PROCEEDINGS OF THE

ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2011

`http://csclp2011.cs.st-andrews.ac.uk`

April 12th-13th, 2011

Editors

Alan M. Frisch
University of York, United Kingdom
`frisch@cs.york.ac.uk`

Barry O'Sullivan
Cork Constraint Computation Centre
University College Cork, Ireland
`b.osullivan@cs.ucc.ie`

THE UNIVERSITY of York

ERCIM
European Research Consortium for Informatics and Mathematics

Preface

This volume contains the invited talk abstracts and the technical papers that were presented at the 2011 ERCIM Workshop on Constraint Solving and Constraint Logic Programming held on April 12th and 13th at Kings Manor, University of York, UK. This event was run on behalf of the ERCIM Working Group on Constraints[1]. ERCIM, the European Research Consortium for Informatics and Mathematics, aims to foster collaborative work within the European research community and to increase co-operation with European industry. Leading research institutes from countries across Europe are members of ERCIM. The ERCIM Constraints working group aims to bring together ERCIM researchers that are involved in research on the subject of constraint programming and related areas.

Constraints have recently emerged as a research area that combines researchers from a number of fields, including Artificial Intelligence, Programming Languages, Symbolic Computing and Computational Logic. Constraint networks and constraint satisfaction problems have been studied in Artificial Intelligence since the 1970s. Systematic use of constraints in programming emerged in the 1980s. The constraint programming process involves the generation of requirements (constraints) and the solution of these requirements, by specialised constraint solvers. Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics (to express geometric coherence in the case of scene analysis), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), operations research problems (like optimization problems), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts), etc. Current research in this area deals with various foundational issues, with implementation aspects and with new applications of constraint programming. The concept of constraint solving forms the central aspect of this research.

This year's workshop programme included invited talks from Holger Hoos (University of British Columbia, Canada) and Torsten Schaub (Universität Potsdam, Germany). the abstracts of which are included here. The main technical programme comprised talks from many constraints researchers on current aspects of their research agenda.

We would like to thank those people who helped bring this workshop to fruition. Peter Nightingale (University of St. Andrews, UK) served as Publicity Chair and promoted the workshop extensively. Becky Polson (University of York, UK) expertly handled many apspects of the workshop organisation, far too many to list. The striking cover of this proceedings was designed by Emma Hodgson (University of York). Finally, we sincerely thank the authors of papers, the speakers and the workshop particpants for such an interesting and engaging programme.

Alan M. Frisch, University of York, United Kingdom
Barry O'Sullivan, University College Cork, Ireland

[1] `http://wiki.ercim.org/wg/Constraints`

Workshop Organisation

Workshop Chairs

Alan M. Frisch, University of York, UK
Barry O'Sullivan, University College Cork, Ireland

Local Arrangements

Alan M. Frisch, University of York, UK

Publicity Chair

Peter Nightingale, University of St. Andrews, UK

Administrative Support

Becky Polson, University of York, UK

Invited Speakers

Holger Hoos, University of British Columbia, Canada
Torsten Schaub, Universität Potsdam, Germany

Supporting Institution

University of York, UK

Sponsors

Association for Constraint Programming
ERCIM Working Group on Constraints

Table of Contents

Programming by Optimisation: Towards a New Paradigm for Developing High-Performance Software

Holger H. Hoos

Department of Computer Science, University of British Columbia, Canada
hoos@cs.ubc.ca

Abstract

When creating software—and in particular, heuristic procedures for solving NP-hard problems—developers frequently explore various ways of achieving certain tasks. Often, these alternatives are eliminated or abandoned early in the process, based on the idea that the flexibility afforded by them would be difficult or impossible to exploit later. In this talk, I challenge this view and advocate an approach that encourages developers to not only avoid premature commitment to certain design choices, but to actively develop promising alternatives for parts of the design.

In this approach, dubbed Programming by Optimisation (PbO), developers specify a potentially large design space of programs that accomplish a given tasks, from which then versions of the program optimised for various use contexts are obtained automatically; per-instance selectors and parallel portfolios of programs can be obtained from the same sources. Using PbO, human experts can focus on the creative task of thinking about possible mechanisms for solving given problems or subproblems, while the tedious task of determining what works best in a given use context is performed automatically, substituting human labour by computation. PbO provides an attractive way of creating software whose performance can be effectively adapted to a wide range of use contexts; it also enables principled empirical investigations into the impact of design choices on performance, into the interaction between design choices and into the suitability of design choices in specific use contexts.

Answer Set Programming: The Solving Paradigm for Knowledge Representation and Reasoning

Torsten Schaub*

Institut für Informatik, Universität Potsdam, Germany
torsten@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP; [5, 6, 1, 4]) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capacities. ASP is particularly suited for modeling problems in the area of knowledge representation and reasoning involving incomplete, inconsistent, and changing information. From a formal perspective, ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT). Applications of ASP include automatic synthesis of multiprocessor systems, decision support systems for NASA shuttle controllers, reasoning tools in systems biology, and many more. The versatility of ASP is also reflected by the ASP solver `clasp` [2, 3, 7], developed at the University of Potsdam, and winning first places at ASP'09, PB'09, and SAT'09.

The talk will give an overview of ASP, its modeling language, solving methodology, and portray some of its applications.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007) 386–392
3. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver clasp: Progress report. Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). Springer (2009) 509–514
4. Gelfond, M.: Answer sets. In Lifschitz, V., van Hermelen, F., Porter, B., eds.: Handbook of Knowledge Representation. Elsevier (2008) 285–316
5. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88), The MIT Press (1988) 1070–1080
6. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
7. Potassco, the Potsdam Answer Set Solving Collection. `http://potassco.sourceforge.net/`

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

Extensible Automated Constraint Modelling

Ozgur Akgun[1], Ian Miguel[1], Alan M Frisch[2], Brahim Hnich[3], and Christopher Jefferson[1]

[1] University of St. Andrews
[2] University of York
[3] Izmir University of Economics

Abstract. In constraint solving, an important bottleneck is the formulation of an effective constraint model of an input problem. The CONJURE system described in this paper, a substantial step forward over prototype versions of CONJURE previously reported, makes a valuable contribution to the automation of constraint modelling by automatically producing constraint models from their specifications in the abstract constraint specification language ESSENCE. A set of rules is used to *refine* an abstract specification into a concrete constraint model. We demonstrate that this set of rules is readily extensible to increase the space of possible constraint models CONJURE can produce. Our empirical results confirm that CONJURE can reproduce successfully the kernels of the constraint models of several benchmark problems found in the literature.

1 Introduction

Automating constraint modelling (the *modelling bottleneck*) is one of the key challenges facing the constraints field [39], and one of the principal obstacles preventing widespread adoption of constraint solving. Without help, it is very difficult for a novice user to formulate an effective (or even correct) model of a given problem. Recently, a variety of approaches have been taken to automate aspects of constraint modelling, including: machine learning [1]; case-based reasoning [30]; theorem proving [5]; automated transformation of medium-level solver-independent constraint models [42, 35, 48, 33]; and refinement of abstract constraint specifications [12] in languages such as ESRA [11], ESSENCE [13], $\mathcal{F}$ [23] or Zinc [31, 29].

We focus on the refinement-based approach, in which a user writes *abstract* constraint specifications that describe a problem above the level at which modelling decisions are made. Abstract constraint specification languages, such as ESSENCE or Zinc support abstract decision variables with types such as set, multiset, function, and relation, as well as *nested* types, such as set of sets, multiset of function variables. Problems can typically be specified very concisely in this way, as demonstrated by the example in Fig. 1. However, existing constraint solvers do not support these abstract decision variables directly, so abstract constraint specifications must be *refined* into concrete constraint models.

CONJURE was introduced in prototype form by Frisch *et al.* [12]. It was able to refine a fragment of ESSENCE limited to nested set and multiset decision variables into models in ESSENCE$'$: a solver-independent modelling language. Frisch and Martinez-Hernandez [32] developed this work further, considering issues involved in channelling

```
given     co,ca: int
letting   item be new type of size co
letting   nat be domain int(1..)
given     vol,val: function (total) item → nat
find      x: set of item
maximising ∑ i : x . val(i)
such that (∑ i : x . vol(i)) ≤ ca
```

Fig. 1: The knapsack problem, given in ESSENCE

efficiently among different representations of abstract variables. The subject of this paper is CONJURE 1.0, a major step forward over the previously reported prototypes (henceforth CONJURE 0.X). We summarise its contributions:

Coverage of ESSENCE**:** CONJURE 1.0 is able to refine all ESSENCE specifications. Whereas CONJURE 0.X demonstrated that refinement could handle types with unbounded nesting, CONJURE 1.0 confirms that refinement can handle all features of ESSENCE.

Extensible Domain-specific Rule Language: The rules used by CONJURE 1.0 are expressed in a new domain-specific declarative language. In contrast each CONJURE 0.X rule was implemented by a piece of Haskell code. This provision separates the tasks of implementation and rule writing and, as we demonstrate, allows the straightforward addition of new rules to extend the space of possible constraint models CONJURE can produce.

Refinement at a Finer Grain: CONJURE 0.X performed refinement at the level of an individual constraint. This is inflexible and inflates the size of the rule base unnecessarily. Moreover, it requires a specification to be flattened prior to refinement. As we will show, this process can mask the structure of the original specification and lead to weaker models. By contrast, CONJURE 1.0 works at the level of expressions, which both reduces the number of rules required and allows us to exploit specification structure to the benefit of the models generated.

Extensive Evaluation: Our working hypothesis is that the kernels[4] of constraint models generated by experts can be automatically generated by refining a problem's specification. The much broader coverage of ESSENCE afforded by CONJURE 1.0 has allowed us to test this hypothesis in much greater detail than previously.

2 Background

ESSENCE [13] is a language for specifying combinatorial (decision or optimisation) problems. It has a high level of abstraction to allow users to specify problems *without* making constraint modelling decisions, supporting decision variables whose types match the combinatorial objects problems typically ask us to find, such as: sets, multisets, functions, relations and partitions. First introduced by the language is its support for the *nesting* of these types, allowing decision variables of type set of sets, multiset of

[4] By which we mean to exclude advanced features of models, such as symmetry breaking and implied constraints.

sets of functions, *etc*. Hence, problems such as the Social Golfers Problem [20], which is naturally conceived of as finding a set of partitions of golfers subject to some constraints, can be specified directly without the need to model the sets or partitions as matrices.

An ESSENCE specification (see [13] for full details), such as that in Fig. 1, identifies: the parameters of the problem class (`given`), whose values are input to specify the instance of the class; the combinatorial objects to be found (`find`); and the constraints the objects must satisfy to be a solution (`such that`). An objective function may also be specified (`min/maximising`) and, for concision, identifiers may be declared (`letting`).

Today's constraint solvers typically support decision variables with atomic types, such as integer or Boolean, have limited support for more complex types like sets or multisets, and no support for nested complex types. Hence, abstract specifications are *refined* by modelling abstract decision variables as constrained collections of variables of more primitive type. CONJURE 1.0, like CONJURE 0.X, employs a system of rules to refine ESSENCE specifications into constraint models in ESSENCE′ [42], a language derived from ESSENCE mainly by removing facilities for abstraction and adding facilities common to existing constraint solvers and toolkits. From ESSENCE′ a tool such as TAILOR [42] can be used to translate the model into the format required for a particular constraint solver.

An abstract specification typically has many constraint models. CONJURE is intended to generate these alternatives by providing multiple refinement rules for each abstract type, corresponding to the various ways in which a decision variable of that type can be modelled. Furthermore, for each way of modelling the decision variables there can be multiple rules to generate alternative models for a constraint on those variables. Consequently, CONJURE often generates many alternative models for an input specification. We aim to encode each rule that for some problem is used in the generation of some good (or perhaps reasonable) model. Given a problem specification and a set of rules the system generates all possible models. If we have encoded a sufficient set of rules, then the kernels of all good (or reasonable) models of the problem should be contained within the set of models. In future, we will investigate restricting this set to *good* models and the selection of either one recommended model or a portfolio of models with complementary strengths.

3 The CONJURE 1.0 Architecture

CONJURE 1.0 is structured like a compiler. Its pipeline starts with parsing, validating the input, and type-checking. After these foundation phases, it prepares the input specification for, and performs, refinement, and does some housekeeping:

1. Parsing
2. Validation
   - *Are all identifiers defined?*
   - *Check consistency of declarations. e.g. a function variable cannot be declared both total and partial.*
3. Type Checking

4. Refinement
5. Model Presentation

Phases 1–3 are foundational, while Phase 5 aids perspicuity. Phase 4 is the core of the refinement process, and is the focus of the remainder of the paper. It consists of multiple reentrant levels: following each rule application the process returns to Phase 4i) in case the result of the rule requires the attention of any of the other levels. We summarise Phase 4:

4i) Partial Evaluation CONJURE 1.0 contains a partial evaluator for ESSENCE. This not only simplifies the output models, but also saves the system from applying rules to expressions that can readily be evaluated.

4ii) Representation Selection Refinement of an abstract expression depends crucially on the representation of the abstract decision variables it involves. Hence, it is natural[5] to select decision variable representations first. This also simplifies the generation of channelling constraints (Level iii) considerably. Typically *structural constraints* are added to the variables in the concrete representation to ensure that the abstract variable is properly represented.

4iii) Auto-Channelling When an abstract decision variable appears in multiple constraints, it can also have multiple representations in a single model (to suit each constraint), in which case *channelling* constraints [6] are necessary to maintain consistency among these different representations. Following [22], channelling constraints are generated simply by constraining the different representations of each abstract variable so that they represent the same abstract object. The resultant equality constraints are refined in the same way as any other constraint in the specification.

4iv) Expression Refinement Having decided on the representation of each abstract decision variable, it remains to refine the expressions that contain them.

In order to produce multiple models, refinement branches in two places in Phase 4: Representation Selection and Expression Refinement. Depending on the rules available in the rule base, each abstract decision variable and each expression can be refined in several different ways.

4 A Rule Language for Refinement

To represent the rules at the heart of CONJURE 1.0, we designed a new domain-specific language[6] that provides all and only the facilities we require, rather than use a general system like Cadmium [8]. CONJURE is run by inputting an ESSENCE specification and a set of rules. This arrangement facilitates the straightforward addition of new rules to extend the space of models CONJURE can produce, as the next section demonstrates.

All CONJURE rules adhere to a single template:

[5] Although in contrast with CONJURE 0.X.

[6] As noted, CONJURE 0.X's rules were written as fragments of Haskell. Some primitive rules, such as those dealing with partial evaluation, are built into CONJURE 1.0 for efficiency.

```
<pattern> [⇝ <output>]* [where <guards>]
          [letting <local identifiers>]
```

A rule matches against `pattern`, producing one or more `outputs` provided the `guards` are satisfied. Local identifiers are used for concision and to identify new variables created by the refinement process. Based upon this template, the CONJURE rule base contains three types of rule:

Representation Selection Rule labels an occurrence of an abstract decision variable in a constraint expression with one of its possible refinements, and adds structural constraints to the model associated with that refinement.

Vertical Rule rewrites an expression, *reducing* its level of abstraction. The expression is rewritten with respect to the labels associated with its constituent variables by the representation selection rules.

Horizontal Rule rewrites an expression *without* changing its level of abstraction. These rules are used to reduce the number of vertical rules required.

We will explain and illustrate the operation of each of the three types by means of an example involving the refinement of the following simple ESSENCE specification:

```
given     x,y : int(0..9)
find      f : function (total) int(0..9) → int(0..9)
such that f(x) = y, |preimage(f,x)| = y
```

This specification contains a single abstract decision variable, a total function `f` mapping from digits to digits. `f` is constrained such that both the image and the cardinality of the inverse of parameter `x` are equal to parameter `y`.

This is a trivial problem, but it serves to illustrate the refinement process. After parsing, validation and type checking, CONJURE arrives at the refinement phase. Representation selection rules are the first to be applied, in Phase 4ii. Consider the following pair of representation selection rules for total functions, which encode one- and two-dimensional matrix representations of a function respectively.

```
function (total) fr → to
    ⇝ Matrix1D
    ⇝ matrix indexed by [fr] of to
    where fr :: int

function (total) fr → to
    ⇝ Matrix2D
    ⇝ matrix indexed by [fr,to] of bool
    ⇝ ∀ i : fr . (∑ j : to . refn[i,j] = 1)
    where fr :: int, to :: int
```

These rules output the identifiers for the representation (`Matrix1D` and `Matrix2D`), the variables in the representation and the structural constraints on these variables, if any. The operator `refn` returns the refined version of the abstract variable to which the vertical rule is applied.

The 1-d matrix is indexed by the domain of the original function whose elements are decision variables with domain equal to the range of the original function. Hence, `f(i)=v` is represented by assigning the `i`th element of the matrix `v`. Since the matrices supported by ESSENCE′ are indexed by integers, there is a guard requiring that the function has an integer domain. There is no such guard on the range of the function — it might have an arbitrarily complex type, in which case the 1-d matrix will be further refined.

The 2-d matrix is indexed by the domain and range of the original function, hence both are required to have an integer domain. Each element of the matrix is a Boolean variable and `f(i) = v` is represented as `m[i,v] = true`. This rule has a third output, a structural constraint that ensures the function is total. This is not necessary in the 1-d representation because each variable must be assigned a value.

In the constraints in the specification, there are two occurrences of `f`. Either of the two rules above may fire on either occurrence, hence refinement branches four ways. We will focus on a single branch in which the 1-d rule has fired on the occurrence in the first constraint, and the 2-d rule has fired on the occurrence in the second. Since the same abstract variable is being represented in two different ways, a channelling constraint is required, and is added in Level iii, giving the intermediate specification:

```
given x,y: int(0..9)
find  f : function (total) int(0..9) → int(0..9)
find  f1: matrix indexed by [int(0..9)] of int(0..9)
find  f2: matrix indexed by [int(0..9),int(0..9)] of bool
such that
  f_Matrix1D(x) = y,
  |preimage(f_Matrix2D,x)| = y,
  ∀ i: int(0..9) . ((∑ j: int(0..9) . f2[i,j]) = 1),
  f_Matrix1D = f_Matrix2D
```

Occurrences of `f` in unrefined expressions are labelled with their chosen representation. The vertical rules use the labels to refine these expressions with respect to their chosen representations. To illustrate, consider this vertical rule for function application, where the 1-d matrix representation has been selected for the function:

```
f(i) ⇝ refn(f)[i]
  where f :: function, repr(f) = Matrix1D
```

Here, `refn(f_Matrix1D)` returns `f1`. Similarly, `repr(f)` returns the identifier for the representation chosen, corresponding to the first output of the representation selection rules. Following the application of the above rule, the first constraint is rewritten to:

```
f1[x] = y
```

We now consider an example of a horizontal rule:

```
|s| ⇝ ∑ i : s . 1 where s :: set
```

This rule replaces the cardinality operator with a summation counting the elements in the set, eliminating the need for a dedicated vertical rule for cardinality. Following its application, the second constraint becomes:

```
∑ i : preimage(f_Matrix2D,x) . 1 = y,
```

The following is a vertical rule to refine quantification over inverse function application:

```
∑ i : preimage(f,j) . k ⇝ ∑ i : r . m[i,j] * k
    where   atomic(f), f :: function, repr(f) = Matrix2D
    letting m be refn(f), r be indices(m)[1]
```

Note the introduction of local identifiers for concision. This rule operates in the context of the selection of the `Matrix2D` representation, transforming the sum over the elements of the inverse of the function into a summation over the indices of the matrix. Following the application of this rule, the second constraint becomes:

```
∑ i : int(0..9) . f2[i,x] = y
```

Partial evaluation has removed the redundant `1`. Space precludes showing the refinement of the channelling constraint, the final unrefined constraint, but it proceeds similarly, giving the model:

```
given x,y: int(0..9)
find f1: matrix indexed by [int(0..9)] of int(0..9)
find f2: matrix indexed by [int(0..9),int(0..9)] of bool
such that
  f1[x] = y,
  (∑ i: int(0..9) . f2[i,x] = y,
  ∀ i: int(0..9) . ((∑ j: int(0..9) . f2[i,j]) = 1)
  ∀ i: int(0..9) . (f1[i] = (∑ j: int(0..9) . f2[i,j])),
```

CONJURE automatically removes the `find f` statement when `f` no longer occurs in any of the constraints.

5 The CONJURE Rule Base and Its Extension

Refining ESSENCE specifications is a complex task. In order to cover the entire ESSENCE language, CONJURE must be able to cope with arbitrarily nested expressions of decision variables with arbitrarily nested abstract types. In order to deal with this complexity, CONJURE 0.X resorted to *flattening* an input specification: decomposing nested expressions into atomic constraints through the introduction of auxiliary variables. A refinement rule was then provided for each flat constraint (for the fragment of ESSENCE that CONJURE 0.X considered). However, the flattening process can mask the structure of the original specification and lead to weaker models as the following example demonstrates. Consider the constraint: `(a ∪ b) ⊆ (c ∩ d)`.

Flattening this constraint gives the constraints `aux0 = a ∪ b`, `aux1 = c ∩ d`, `aux0 ⊆ aux1` containing two auxiliary set variables. This is expensive: each auxiliary set variable incurs the cost of a set of concrete variables and constraints to represent it. This cost increases considerably if the sets have nested types. If we do not flatten, we can use horizontal rules to rewrite the expression to a far more efficient pair of quantified expressions:

```
∀ i : a . (i ∈ c ∧ i ∈ d),  ∀ i : b . (i ∈ c ∧ e ∈ d)
```

As we have seen, therefore, CONJURE 1.0 refines at a *finer grain* than CONJURE 0.X, refining expressions rather than entire constraints, hence eliminating the need for flattening. Even so, a large number of rules are ostensibly required to refine all of ESSENCE. The key to reducing this number are the horizontal rules, which rewrite

```
set of τ                                           [Representation selection]
  ⇝ Gent
  ⇝ matrix indexed by [r] of int(0..n)
  ⇝ ∀ i : r . ((refn[i] = 0) ∨
      (refn[i] = 1 +
         (∑ j : int(..i-1) ∩ r . (refn[j] > 0))))
  where   τ :: int
  letting r be dom(τ), n be domSize(τ)
```

```
∀ i : s . k                                                      [Vertical]
  ⇝ ∀ i : r . (m[i] > 0) ⇒ k
  where   atomic(s), s :: set, repr(s) = Gent
  letting m be refn(s), r be indices(m)[0]
```

```
∃ i : s . k                                                      [Vertical]
  ⇝ ∃ i : r . (m[i] > 0) ∧ k
  where   atomic(s), s :: set, repr(s) = Gent
  letting m be refn(s), r be indices(m)[0]
```

```
∑ i : s . k                                                      [Vertical]
  ⇝ ∑ i : r . (m[i] > 0) * k
  where   atomic(s), s :: set, repr(s) = Gent
  letting m be refn(s), r be indices(m)[0]
```

Fig. 2: The four refinement rules sufficient to integrate the Gent representation of sets into CONJURE.

expressions into a form for which we have a vertical rule, as per the cardinality horizontal rule in the previous section. Horizontal rules are *representation independent* — a horizontal rewrite of an expression is valid irrespective of the concrete representation of its constituent variables. Hence, the horizontal rules reduce the set of vertical rules required and the set of horizontal rules does not grow as representations are added.

Using horizontal rules we have been able to reduce the set of rules required for each of ESSENCE's types to a minimum. For each representation of an abstract type, a representation selection rule is required as well as a small set of vertical rules. For set, multiset, relation, and partition variables, one vertical rule per quantifier is necessary. For functions, vertical rules are required for function application, quantification over the inverse of the function and quantification over the 'defined' elements of a partial function.

Consequently, at present CONJURE 1.0 has 37 representation selection rules, and 96 vertical and horizontal rules.

The use of horizontal rules to keep vertical rules to a minimum also reduces the effort of adding new representations to the system. To illustrate, we consider adding rules to support the Gent representation of sets [27]. The properties of this representation are irrelevant — this example is chosen as being a non-trivial, unusual representation of a set typical of a representation one might wish to add to those already present in the rule base. In brief, a set S of integers drawn from some range d is represented by a matrix g, indexed by d, of decision variables with domain $0..n$, where n is either the fixed or the maximum cardinality of S. Element i is in S iff $g[i]$ is non-zero. Furthermore, the non-zero elements of g are required to be in ascending order. If, for example, $S = \{1, 2, 4\}$, drawn from $1..5$, then $g = [1, 2, 0, 3, 0]$. Just four rules, in addition to the existing horizontal rules, are necessary to integrate the Gent representation fully into CONJURE (Fig.2), such that any of the set operators available in ESSENCE can be refined.

6 Evaluation

A milestone represented by CONJURE 1.0 is full coverage of ESSENCE: it has at least one variable representation rule for every abstract variable type, and rules for the operators defined on them. We now test the hypothesis that the kernels of constraint models written by experts can be automatically generated by refining a problem's specification. We wrote specifications for a diverse set of 32 benchmark problems drawn from the literature and refined them with CONJURE. Table 1 presents the results: the number of generated models, papers that contain a kernel we generate and the abstract parameters and variables involved in the problem. Papers containing n kernels we generate are labelled $\times n$. Notice the variety of decision variable types involved in the benchmark problems, representing a proof that the current collection of rules, the rewrite rule language, and the Conjure system as a whole is capable of refining a variety of abstract problem specifications into concrete models.

The number of models generated for a problem specification depends on the number of representation options for the involved abstract decision variables. For instance, the *Maximum Density Still Life* contains a set decision variable whose elements are tuples and currently the system has only one variable selection rule for this type. Problems such as *Magic Hexagon* only contain decisions variables that are concrete, so do not require refinement. We did find papers containing kernels which we are currently unable to generate, for example for *Langford's Number Problem* and *Maximum Density Still Life*. These come from complex reformulations of the problem. In each of these cases, an alternative Essence specification allows CONJURE to generate the missing kernel.

Further research is necessary to improve the quality of generated models, unsurprisingly since producing a good model is well known to be difficult. We have established that good rewrite rules are applicable to many problems and we hope as as our horizontal rule database increases, we will produce better models for all problems. For example, one common method of iterating over pairs of distinct elements is to use $\forall \mathtt{i}, \mathtt{j} : \mathtt{s}.(\mathtt{i} \neq \mathtt{j}) \Rightarrow \mathtt{k}$. Constraints of this form arise in many problems, including *Golomb ruler* and *Social Golfers*. Our standard rules would refine $\mathtt{i} \neq \mathtt{j}$ using the representation chosen for the elements of s, which can lead to a complex constraint. If s is represented with the explicit representation then $\mathtt{i} \neq \mathtt{j}$ is true if and only if `i` and `j` are represented by different elements of the matrix in the refinement. This leads to the rule:

```
∀ i,j : s . (i ≠ j) ⇒ k ⇝                       [Vertical]
  ∀ i',j' : r . (i' ≠ j') ⇒ k {i → m[i'],j → m[j']}
  where   atomic(s), s :: set, repr(s) = Explicit
  letting m be refn(s), r be indices(m)[0]
```

Table 1: Runnning CONJURE on benchmark problems.

Problem name	Models	Reference	Nb. abstract params and vars
Car Sequencing	128	[18]	*4 functions, 1 relation*
Template Design	16	[38]	*2 function variables, 1 mapping msets to integers*
Low Autocorellation Binary Sequences	4	[17]	*1 function*
Golomb Ruler	81	[47, 37]	*1 set*
All-interval series	8	[7]	*2 functions*
Vessel loading	256	[2]	*9 functions, 1 mapping from a set*
Perfect Square Placement	1024	[3]	*2 functions*
Social Golfers	3	[28, 21]	*multiset of partitions*
Progressive Party	81	[45]	*1 set, 1 set of functions*
Schur's Lemma	81	[10]$\times$2	*1 partition*
Traffic Lights	2	[26]	*1 set of functions mapping integers to tuples*
Magic Squares	1	[40]	*1 2-dimensional matrix*
Bus Driver Scheduling	27	[34]	*1 set of sets, 1 partition*
Magic Hexagon	1	Model from CSPLib 23	*1 2-dimensional matrix*
Langford's Number Problem	32	[25]	*1 function*
Round Robin Tournament Scheduling	27	[16]	*1 relation between 2 integers and 1 set*
BIBD	16	[36]	*1 relation between 2 unnamed types*
Balanced Academic Curriculum Problem	512	[24]	*2 functions, 1 relations*
Rack Configuration Problem	288	[28]	*7 functions, 1 mapping integers to sets*
Maximum Density Still Life	1	[44]	*1 set of tuples*
Word Design for DNA Computing	16	Model from CSPLib 33	*1 set of functions*
Warehouse Location Problem	16	[48]	*3 functions, 1 mapping tuples to integers*
Fixed Length Error Correcting Codes	16	[15]	*2 functions, 1 mapping tuples to integers*
Steel Mill	4	[9]	*3 functions, 1 from sets*
N-Fractions Puzzle	16	[16]	*1 function*
Steiner Triple Systems	9	[28, 21]	*1 set of sets*
N-Queens Problem	4	[25]$\times$2	*1 function*
Peaceably Co-existing Armies of Queens	1	[46]	*1 set of tuples*
Maximum Clique Problem	81	[41]	*1 set, 1 set of sets*
Graph Colouring	4	[19, 4]	*1 function*
SONET Configuration	27	[14][1]	*1 mset of sets, 1 set of sets*
Knapsack Problem	36	[43]	*2 functions, 1 set*

[1]Some models in this paper have set variables, which Conjure currently always refines.

7 Conclusion and Future Work

We have presented the CONJURE 1.0 automated constraint modelling system and demonstrated its ability to reproduce the kernels of the constraint models of 32 benchmark problems found in the literature. It achieves full coverage of the ESSENCE language via a new domain-specific rule language, whose features include: fine-grained refinement to avoid the need for flattening, which, as we have demonstrated, can impair the models produced; horizontal rules that normalise expressions to reduce considerably the total number of rules necessary for refinement; easy extensibility.

In future we of course wish to go beyond model kernels to produce full models of the same quality as those found in the literature, including symmetry breaking and implied constraints. CONJURE's flexible rule-based architecture is ideally placed to achieve these aims in large part by adding new rules to those available (cf. the example in the previous section). Furthermore, we will prune the set of models produced to contain only the most effective models. In part, we plan to achieve this by applying a prioritisation system to rule application. This will allow refinement paths that are provably superior to dominate those shown to be weaker.

References

1. Christian Bessiere, Remi Coletta, Frederic Koriche, and Barry O' Sullivan. Acquiring constraint networks using a SAT-based version space algorithm. In *AAAI 2006*, pages 1565–1568, 2006.
2. K. N. Brown. Loading supply vessels by forward checking and unenforced guillotine cuts. In *17th Workshop of the UK Planning and Scheduling SIG*, 1998.
3. Hadrien Cambazard and Barry O'Sullivan. Propagating the Bin Packing Constraint Using Linear Programming. In *CP 2010*, pages 129–136. 2010.
4. Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers and Mathematics with Applications*, 34(9):1 – 14, 1997.
5. John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In *Proc. of ECAI 2006*, pages 73–77. IOS Press, 2006.
6. B.M.W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.
7. C. Choi and J. Lee. On the pruning behaviour of minimal combined models for permutation csps. In *CP 2002 Workshop on Reformulation*, 2002.
8. Gregory J. Duck, Leslie De Koninck, and Peter J. Stuckey. Cadmium: An implementation of acd term rewriting. In *ICLP*, pages 531–545, 2008.
9. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling: Exploiting common patterns in constraint programming. In *the International Workshop on Reformulating CSPs*, pages 27–41, 2002.
10. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *CP 2002*, pages 462–476, 2002.
11. Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR 2003*, pages 214–232, 2003.
12. A. M. Frisch, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The rules of constraint modelling. In *Proc. of the IJCAI 2005*, pages 109–116, 2005.
13. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints 13(3)*, pages 268–306, 2008.
14. Alan M. Frisch, Brahim Hnich, Ian Miguel, Barbara M. Smith, and Toby Walsh. Transforming and refining abstract constraint specifications. In *6th Symposium on Abstraction, Reformulation and Approximation*, pages 76–91. Springer, 2005.
15. Alan M. Frisch, Christopher Jefferson, and Ian Miguel. Constraints for breaking more row and column symmetries. In *CP 2003*, pages 318–332, 2003.
16. Alan M. Frisch, Christopher Jefferson, and Ian Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In *ECAI 2004*, pages 171–175, 2004.
17. Ian P. Gent and Barbara Smith. Symmetry breaking during search in constraint programming. In *ECAI'2000*, pages 599–603, 1999.
18. M. Gravel, C. Gagné, and W. L. Price. Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem. *Journal of the Operational Research Society*, 56(11):1287–1295, 2005.
19. Jin-Kao Hao and Raphaël Dorne. Empirical studies of heuristic local search for constraint solving. In *CP '96*, pages 194–208. 1996.
20. Warwick Harvey. Symmetry breaking and the social golfer problem. In *Proceedings SymCon-01: Symmetry in Constraints, co-located with CP 2001*, pages 9–16, 2001.
21. Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *J. Artif. Intell. Res. (JAIR)*, 24:109–156, 2005.
22. Bernadette Martínez Hernández and Alan M. Frisch. The automatic generation of redundant representations and channelling constraints. In *Trends in Constraint Programming*, chapter 8, pages 163–182. ISTE, May 2007.
23. Brahim Hnich. Thesis: Function variables for constraint programming. *AI Commun*, 16(2):131–132, 2003.

24. Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Modelling a balanced academic curriculum problem. In *CP-AI-OR-2002*, pages 121–131, 2002.
25. Brahim Hnich, Barbara M. Smith, and Toby Walsh. Dual modelling of permutation and injection problems. *J. Artif. Int. Res. (JAIR)*, 21:357–391, February 2004.
26. Walter Hower. Revisiting global constraint satisfaction. *Information Processing Letters*, 66(1):41–48, 1998.
27. Christopher Jefferson. *Thesis: Representations in Constraint Programming*. PhD thesis, University of York, 2007.
28. Zeynep Kiziltan and Braham Hnich. Symmetry breaking in a rack configuration problem. In *the IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints*, 2001.
29. Leslie De Koninck, Sebastian Brand, and Peter J. Stuckey. Data independent type reduction for zinc. In *ModRef10*, 2010.
30. James Little, Cormac Gebruers, Derek G. Bridge, and Eugene C. Freuder. Using case-based reasoning to write constraint programs. In *CP*, page 983, 2003.
31. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints 13(3)*, 2008.
32. Bernadette Martínez-Hernández. *Thesis: The Systematic Generation of Channelled Models in Constraint Satisfaction*. PhD thesis, University of York, 2008.
33. P. Mills, E.P.K. Tsang, R. Williams, J. Ford, and J. Borrett. EaCL 1.5: An easy abstract constraint optimisation programming language. Technical report, University of Essex, Colchester, UK, December 1999.
34. Tobias Muller. Solving set partitioning problems with constraint programming. In *PAPPACT98*, 1998.
35. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Proc. of CP 2007*, pages 529–543, 2007.
36. K. Petrie. *Constraint Programming, Search and Symmetry*. PhD thesis, University of Huddersfield, 2005.
37. S. Prestwich. Negative effects of modeling techniques on search performance. *Annals of Operations Research*, 118:137–150, 2003. 10.1023/A:1021809724362.
38. Les Proll and Barbara Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS J. on Computing*, 10:265–275, March 1998.
39. Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In *Principles and Practice of Constraint Programming - CP 2004*, pages 5–8, 2004.
40. Philippe Refalo. Impact-based search strategies for constraint programming. In *CP 2004*, volume 3258, pages 557–571. 2004.
41. Jean-Charles Regin. Using constraint programming to solve the maximum clique problem. In *CP 2003*, pages 634–648, 2003.
42. Andrea Rendl. *Thesis: Effective Compilation of Constraint Models*. PhD thesis, University of St. Andrews, 2010.
43. Meinolf Sellmann. Approximated consistency for the automatic recording constraint. *Comput. Oper. Res.*, 36:2341–2347, August 2009.
44. Barbara Smith. A dual graph translation of a problem in life. In *CP 2002*, volume 2470, pages 89–94. Springer Berlin / Heidelberg, 2006.
45. Barbara M. Smith, Sally C. Brailsford, Peter M. Hubbard, and H. Paul Williams. The progressive party problem: Integer linear programming and constraint programming compared. In *CP '95*, pages 36–52, 1995.
46. Barbara M. Smith, Karen E. Petrie, and Ian P. Gent. Models and symmetry breaking for peaceable armies of queens. In *Integration of AI and OR Techniques in CP for COP*, pages 271–286. 2004.
47. Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *17th National Conference on AI*, pages 182–187. AAAI Press, 2000.
48. Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, USA, 1999.

The Open Stacks Problem

An automated modelling case study

Ozgur Akgun, Ian Miguel, Christopher Jefferson
{ozgur,ianm,caj}@cs.st-andrews.ac.uk

School of Computer Science, University of St. Andrews, UK

Abstract. This paper presents a case study of automated modelling using ESSENCE and CONJURE. The problem we study is the open stacks problem[5]. We start with a naive problem specification and show how CONJURE generates a selection of constraint programming models automatically. After observing the results, we modify the original problem specification to generate better models. Finally, we further improve the generated models by introducing a new representation for partial and injective function variables. The newly added representation is useful for not only this specific problem but also any other problem with a similar structure.

1 Introduction

This paper presents a case study of automated modelling using CONJURE. CONJURE is an automated modelling tool whose inputs are a high level problem specification given in ESSENCE[2], and a collection of refinement rules. It performs problem class level transformations to generate multiple ESSENCE′[4] models for the input problem specification.

Often referred to as the *modelling bottleneck*, correctly modelling a given problem is not a trivial task. Furthermore, producing a *good* model for a given problem is an active area of research. Even for constraint modelling experts, producing a good model remains an art rather than a science – we are far from fully understanding when a given model is better than another one.

In contrast to many existing languages for constraint modelling which only support decision variables of primitive types (boolean and integer), ESSENCE provides complex types for decision variables and parameters to ease modelling. In order to tackle the challenge of producing good models, CONJURE takes a rather unique approach. The applied transformations are extensible by design, to be able to capture new modelling idioms as they are discovered by human experts. Furthermore, the objective of the system is to generate all possible models using the available refinement rules. The output is not a single random model, instead it is a portfolio of models ready for further investigation.

The typical use case for a problem owner doesn't require any alterations to the refinement rules. The existing refinement rules are capable of refining all types and expressions found in ESSENCE to generate valid models. Refinement rule authoring is only required when a new variable representation or a new transformation is discovered.

The rest of the paper is structured as follows. We first give a precise description of the open stacks problem together with a simple example. In section 2, we give the architecture of the CONJURE system. Section 3 gives a precise problem specification and demonstrates how the system operates on it with the help of some example refinement rules. Section 4 builds on an observation specific to the problem at hand, and improves the problem specification. Finally, we compare the two specifications and conclude.

1.1 The problem

Taken from the Constraint Modelling Challenge 2005 proceedings[5]:

> A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customers order is started (i.e. the first product in the order is being made) a stack is created for that customer. [Each customer places exactly one order.] When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the maximum number of stacks that are in use simultaneously, i.e. the number of customer orders that are in simultaneous production, should be minimised.

For clarity, we introduce a simple example consisting of 5 products and 5 customers. In Fig 1, the demand matrix, D, is given. Every row in this matrix correspond to a single customer, and every column correspond to a single product. D_{ij} is 1 iff c_i placed an order for p_j.

	p_1	p_2	p_3	p_4	p_5
c_1	1	1	0	1	0
c_2	0	1	0	1	1
c_3	0	0	1	1	0
c_4	0	0	1	0	0
c_5	0	0	1	0	0

Fig. 1. D, the demand matrix

	p_1	p_2	p_3	p_4	p_5
c_1	1	1	1	1	0
c_2	0	1	1	1	1
c_3	0	0	1	1	0
c_4	0	0	1	0	0
c_5	0	0	1	0	0

Fig. 2. O, the open stacks matrix

In order to calculate the number of stacks needed for this instance, we use an intermediate matrix of boolean variables, the open stacks matrix shown in Fig 2. O_{ij} is 1 iff a stack for c_i is open at the time of producing p_j.

Using this example and without permuting the order of production, at the time of producing p_3, all 5 stacks are needed. However, as can be seen in Fig 3 changing the order of production to $(p_1, p_2, p_4, p_5, p_3)$ at most 3 stacks are needed at any time.

	p_1	p_2	p_4	p_5	p_3
c_1	1	1	1	0	0
c_2	0	1	1	1	0
c_3	0	0	1	1	1
c_4	0	0	0	0	1
c_5	0	0	0	0	1

Fig. 3. The optimal solution using only 3 stacks

2 The architecture of Conjure

Conjure 1.0 is structured like a compiler. It operates by applying the given refinement rules to the input problem specification in several reentrant phases. The refinement rules are annotated by a precedence level; rules with higher precedence are always applied before the others. The rules residing at the same precedence level are applied simultaneously to generate multiple output models. Once a rule is applied, the process continues from the highest precedence level.

There are two main kinds of refinement rules, rules to select a concrete representation for an Essence type, and rules to transform Essence expressions.

The pipeline starts with parsing, validating the input, and type-checking. After these foundation phases, it prepares the input specification for, and performs, refinement, and does some housekeeping:

1. Parsing
2. Validation
   - *Are all identifiers defined?*
   - *Check consistency of declarations. e.g. a function variable cannot be declared both total and partial.*
3. Type Checking
4. Refinement
5. Model Presentation

Phases 1–3 are foundational, while Phase 5 aids perspicuity. Phase 4 is the core of the refinement process, and is briefly discussed below. It consists of multiple reentrant levels: following each rule application the process returns to Phase 4i) in case the result of the rule requires the attention of any of the other levels. We summarise Phase 4:

4i) Partial Evaluation Conjure 1.0 contains a partial evaluator for Essence. This not only simplifies the output models, but also saves the system from applying rules to expressions that can readily be evaluated.

4ii) Representation Selection Refinement of an abstract expression depends crucially on the representation of the abstract decision variables it involves. Hence, it is natural to select decision variable representations first. This also simplifies the generation of channelling constraints (Level iii) considerably. Typically *structural constraints* are added to the variables in the concrete representation to ensure that the abstract variable is properly represented.

4iii) Auto-Channelling When an abstract decision variable appears in multiple constraints, it can also have multiple representations in a single model (to suit each constraint), in which case *channelling* constraints [1] are necessary to maintain consistency among these different representations. Following [3], channelling constraints are generated simply by constraining the different representations of each abstract variable so that they represent the same abstract object. The resultant equality constraints are refined in the same way as any other constraint in the specification.

4iv) Expression Refinement Having decided on the representation of each abstract decision variable, it remains to refine the expressions that contain them.

In order to produce multiple models, refinement branches in two places in Phase 4: Representation Selection and Expression Refinement. Depending on the rules available in the rule base, each abstract decision variable and each expression can be refined in several different ways.

3 A *precise* problem description

In this section we give a precise problem description, first in English and then in ESSENCE.

> Given a number of customers, a number of products, and a demand relation between customers and products, find a permutation of products, such that if production is done in this order the number of stacks needed at any time is minimised. A stack is opened for every customer when the first product they demand is produced and it is closed when there are no more products for them to be produced.

The ESSENCE problem specification (Fig 4) is very close to the English description and should be readily understandable. The problem is parameterised over two integers and one 2 component relation. Three letting statements are used to bind names to commonly used domains. Notice, two of these names, *PRODUCT* and *TIMESLOT*, are actually bound to the same domain; different names are introduced to capture the different meanings and ease understanding in the rest of the problem specification.

The decision central to the problem is finding a permutation of the given products, hence the function variable named *timeof*. It is decorated with two attributes; *total* to ensure every value in the domain set is assigned to a value from the range set, and *injective* to ensure that the function preserves distinctness.

The function variable named *stackOpen*, mapping every customer to a pair of time slots, is used to mark the stack opening and closing times for every customer. The relation variable *isOpenStack* is only used to ease the calculation of *nbStacks*, the maximum number of stacks needed at any time slot.

First two constraints, lines 16 and 17 in Fig 4, constrain a stack for a customer to remain open for the duration of all the demand points for that customer. The

```
language Essence 2.0

given   nbProducts, nbCustomers: int(1..)
letting PRODUCT  be domain int(1..nbProducts),
        TIMESLOT be domain int(1..nbProducts),
        CUSTOMER be domain int(1..nbCustomers)
given   demand      : relation of (CUSTOMER * PRODUCT)
find    timeof      : function (total, injective) PRODUCT → TIMESLOT
find    stackOpen   : function (total) CUSTOMER → tuple (TIMESLOT,TIMESLOT)
find    isOpenStack : relation of (CUSTOMER * PRODUCT)
find    nbStacks    : PRODUCT

minimising nbStacks

such that
    forall (c,p) : demand . stackOpen(c)[0] ≤ timeof(p),
    forall (c,p) : demand . stackOpen(c)[1] ≥ timeof(p),

    forall c : CUSTOMER . forall t : TIMESLOT .
        isOpenStack(c,t) = (stackOpen(c)[0] ≤ t ∧
                            stackOpen(c)[1] ≥ t),

    forall t : TIMESLOT .
        nbStacks ≥ |isOpenStack(_,t)|
```

Fig. 4. Problem specification in Essence

universal quantification over *demand*, gives only those customer–product pairs for which a demand exists. The third constraint simply relates the function variable to the relation variable such that *isOpenStack(c,t)* is true iff a stack for customer *c* is open at time slot *t*.

3.1 Refinement

The process of auto-modelling the given problem specification is explained in this section. There are 3 decision variables and 1 parameter requiring refinement. Conjure rule-base contains 2 refinement options for relation variables, and 2 refinement options for function variables. Using all the existing refinement rules, at least 16 valid models can be generated; some with multiple representations for a single decision variable and with the appropriate channelling constraints in place. Here, we only give those refinement rules necessary to generate one of the resultant models, arguably the best one.

All Conjure rules adhere to a single template:

```
<pattern> [⇝ <output>]*
          [where <guards>]
          [letting <local identifiers>]
```

A rule matches against `pattern`, producing one or more `outputs` provided the `guards` are satisfied. Local identifiers are used for concision and to identify new variables created by the refinement process.

First, we start by giving the necessary representation selection rules. A representation selection rule, matches with an Essence type, and outputs 3 things: the name of the representation selected, the concrete representation in the form of a valid ESSENCE type, and any structural constraints to be added when this rule is applied. Every output is preceded by a $\rightsquigarrow$ sign. The structural constraints component is optional, it can be omitted if the type refinement doesn't imply any new constraints.

```
function (total, injective) a → b ⇝ Function1DMatrix
                                  ⇝ matrix indexed by [a] of b
                                  ⇝ alldifferent(refn)
    where a :: int
```

Fig. 5. Representation selection rule for a total and injective function variable.

Fig 5 gives a representation selection rule for total and injective function variables. A decision variable with a matching type can be refined to a one-dimensional matrix with an *alldifferent* posed on it. Here, *refn* is a special operator which returns the newly created variable after applying this representation selection rule.

```
function (total) a → b ⇝ Function1DMatrix
                       ⇝ matrix indexed by [a] of b
    where a :: int
```

Fig. 6. Representation selection rule for a total function variable.

Fig 6 gives a representation selection rule for total function variables. Similar to Fig 5, a decision variable with a matching type can be refined to a one-dimensional matrix. Notice, we don't pose an *alldifferent* constraint in this case, since the function variable is not marked to be injective.

```
relation of (a * b) ⇝ Relation2D
                    ⇝ matrix indexed by [a,b] of bool
    where a :: int, b :: int
```

Fig. 7. Representation selection rule for a 2-component relation.

Fig 7 gives a representation selection rule for 2-component relation variables. Both the parameter *demand* and decision variable *isOpenStack* will use this representation during refinement.

Fig 8 gives a refinement rule which fires while refining the first two constraints, on lines 16 and 17 in Fig 4. The pattern on the left hand side of the $\rightsquigarrow$ sign, matches with a universal quantification over a 2-component matrix. `k` in the pattern, matches with the body of the quantification. The returned expression contains a guard in the form of an logical implication on `k`, ensuring k is only ever applied for values found in the relation *rel*.

There are some important operators used in this refinement rule:

`(::)` does type checking. It accepts two arguments, and returns *true* if the types of the parameters are same, and *false* otherwise. Each parameter can be an ESSENCE expression or an ESSENCE types.

`repr` returns the selected representation for an atomic expression. It can only be used in the *where* clause of a refinement rule and fails the application of the rule if a representation is not chosen for its argument.

```
forall (i,j) : rel . k
    ⇝ forall i : ri .
         forall j : rj .
            m[i,j] ⇒ k
    where   rel :: relation of (int,int),
            repr(rel) = Relation2D
    letting m be refn(rel),
            ri be indices(m)[0],
            rj be indices(m)[1]
```

Fig. 8. Refinement rule for quantification over a 2-component relation with *Relation2D* as the selected representation.

`refn`	returns the concrete representation for an atomic expression.
`indices`	returns the indexing domains of a matrix. It returns a tuple containing every indexing domain in their respective positions. Individual indices can be projected out of the tuple using the tuple indexing operator `[]`.

```
|rel(_,b)| ⇝ sum (i,j) : rel . (j = b) * rel(i,j)
```

Fig. 9. Horizontal refinement rule for cardinality of relation projection.

This rule given in Fig 9 is used while refining the last constraint, starting on line 23 in Fig 4. It is called a *horizontal* rule, because it doesn't refer to the representation of any decision variable using the *repr* operator, nor it requests the refinement of a decision variable using the *refn* operator. The existence of horizontal rules are very important to the scalability of the rules database. A vertical rule needs to be added for every newly added variable representation, whereas horizontal rules can be used independently of the chosen representation.

Notice, since horizontal rules do not do type refinement, the resultant expression needs further refinement rule applications. In this specific case, a rule very similar to that of Fig 8 will be applied to refine the relation variable to a matrix.

Given refinement rules are sufficient to produce the output model given in Fig 11. Although automatically generated from a problem specification, the model is very close to what a human modeller would write, once the modelling decisions are decided upon. Except the last constraint, starting on line 28, which would have been written as in Fig 10. This is due to using a horizontal rule to transform the cardinality constraint to a nested *sum*. Luckily, in this case and in many other cases where we consider the use of horizontal rules to be beneficial, the two forms are identical modulo instantiation. Any tool which instantiates the problem class model with given parameters will generate exactly the same constraints.

```
    forall t : TIMESLOT .
        nbStacks ≥ sum c : CUSTOMER . isOpenStack[c,t]
```

Fig. 10. The last constraint of Fig 11, slightly modified.

```
language ESSENCE′ 1.0

given nbProducts, nbCustomers : int(1..)
letting PRODUCT  be domain int(1..nbProducts),
        TIMESLOT be domain int(1..nbProducts),
        CUSTOMER be domain int(1..nbCustomers)
given demand      : matrix indexed by [CUSTOMER,PRODUCT] of bool
find timeof       : matrix indexed by [PRODUCT] of TIMESLOT
find stackOpened  : matrix indexed by [CUSTOMER] of TIMESLOT
find stackClosed  : matrix indexed by [CUSTOMER] of TIMESLOT
find isOpenStack  : matrix indexed by [CUSTOMER,TIMESLOT] of bool
find nbStacks     : PRODUCT

minimising nbStacks

such that
    alldifferent(timeof),

    forall c : CUSTOMER . forall p : PRODUCT .
        demand[c,p] ⇒ stackOpened[c] ≤ timeof[p],

    forall c : CUSTOMER . forall p : PRODUCT .
        demand[c,p] ⇒ stackClosed[c] ≥ timeof[p],

    forall c : CUSTOMER . forall t : TIMESLOT .
        isOpenStack[c,t] = ((stackOpened[c] ≤ t) ∧ (t ≤ stackClosed[c])),

    forall t : TIMESLOT .
        nbStacks ≥ sum t2 : TIMESLOT . (
                        (t = t2) *
                        (sum c : CUSTOMER . isOpenStack[c,t])
                  )
```

Fig. 11. The auto-generated model for the problem specification given in Fig 4.

4 Better understanding the problem

As noted in [5], an important observation about the problem enables us to drastically shrink the problem size.

Observation. For every product p, if there exists another product p' such that the set of customers demanding p is a *subset* of the set of customers demanding p', p doesn't need to be considered while sequencing products.

For example, in the instance given at Fig 1, products p_1, p_2 and p_5 do not need to be considered while sequencing the rest of the products. Namely, only sequencing p_3 and p_4 and minimising the required number of stacks for these two products is enough to optimally solve the original problem.

The observation means we can preprocess the data file to remove such products, and use the problem specification we already have. Alternatively, we can encode the property in the problem specification; enabling us to leverage from the observation while still using ESSENCE and staying at the problem class level.

In the light of this observation, instead of finding a total mapping from products to time-slots, we need to search for a partial mapping. To accomplish this, we simply modify the attributes of *timeof* to include *partial* instead of *total*.

```
find timeof : function (partial, injective) PRODUCT → TIMESLOT
```

timeof being partial function means some products will be assigned to time-slots and others won't. Since the function is still *injective*, it preserves distinctness through assigned time-slots.

Two constraints are needed to statically compute those products which need sequencing.

```
forall p1 : PRODUCT . (
   (exists p2 : PRODUCT . (p1 ≠ p2) ∧ (demand(_,p1) ⊆ demand(_,p2)))
   ⇒ (p1 ∉ defined(timeof))
),

forall p1 : PRODUCT . (
   (forall p2 : PRODUCT . (p1 ≠ p2) ⇒ !(demand(_,p1) ⊆ demand(_,p2)))
   ⇒ (p1 ∈ defined(timeof))
),
```

The operator `defined` works on function variables and returns the set of values the function is defined on.

Only the second and third constraints from the original problem specification (Fig 4) need to be modified slightly.

```
forall (c,p) : demand .
   p ∈ defined(timeof) ⇒ stackOpened(c) ≤ timeof(p),

forall (c,p) : demand .
   p ∈ defined(timeof) ⇒ stackClosed(c) ≥ timeof(p),
```

Other constraints remain unchanged.

The changes made on the problem specification arise from a better understanding of the problem. Although this may be perceived as pushing the modelling bottleneck up to the ESSENCE level, it should be noted that the modifications are done at problem specification level. Modelling decisions, such as how to model a partial and injective function variable, do not need to be considered. We improve the problem specification merely by changing how a decision variable is declared.

4.1 Refinement

CONJURE can already refine the modified problem description to valid constraint models. In order to be as generic as possible, the existing refinement uses two 1-dimensional matrices to represent a partial and injective function variable. Using this representation (Fig 12), the constraint to pose distinctness is highly inefficient.

In the specific case of a function variable mapping integers to integers, we can do better. Instead of introducing a boolean variable for every possible mapping, we can introduce a dummy value in the mapped domain, and use *alldifferent_except* to ensure distinct assignments (Fig 13).

```
function (partial, injective) a → b
    ⤳ PartialFunction1DMatrix
    ⤳ matrix indexed by [a] of tuple (bool,b)
    ⤳ forall i,j : a . (
          ((refn[i][0] = true) ∧ (refn[j][0] = true))
          ⇒ (refn[i][1] ≠ refn[j][1])
        )
    where a :: int
```

Fig. 12. Generic representation for partial and injective function variables.

```
function (partial, injective) a → int(1..b)
    ⤳ PartialFunctionIntInt
    ⤳ matrix indexed by [a] of int(0..b)
    ⤳ alldifferent_except(refn,0)
    where a :: int
```

Fig. 13. A specific representation for function variable mapping integers to integers.

This is a very specific variable representation, yet it occurs very often. In the process of modelling the open stacks problem, this specific case is discovered and proved to be highly efficient. Future problems with a similar structure will benefit from it for free.

The refinement rule in Fig 14 is needed to fully refine the modified problem specification using the newly added representation.

```
e ∈ defined(fn) ⤳ refn(fn)[e] > 0
    where repr(fn) = PartialFunctionIntInt
```

Fig. 14. A new refinement rule is necessary to fully refine the problem specification.

The result of refining the modified problem specification is given in Fig 15.

5 Experimental results

We ran experiments on the instances provided by [5]. Search node counts and search times are recorded for over 700 problem instances for both the original and the improved model.

The plots given in Fig 16 and Fig 17 are comparing the two models by the number of search nodes and by actual search time respectively. The instances are ordered by the amount of gain. Both vertical axes are log-scaled.

Comparing constraint models is not a trivial task. Given two problem class models, one can be better than the other for some parameter instantiations and worse for some others.

Number of search nodes used by the underlying solver is one possible measure for comparing two constraint models. It gives a good understanding as long as

```
language ESSENCE' 1.0

given nbProducts : int(1..)
letting PRODUCT be domain int(1..nbProducts)
letting TIMESLOT be domain int(1..nbProducts)

given nbOrders : int(1..)
letting CUSTOMER be domain int(1..nbOrders)

given demand      : matrix indexed by [CUSTOMER,PRODUCT] of bool
find timeof       : matrix indexed by [PRODUCT] of int(0..nbProducts)
find stackOpened  : matrix indexed by [CUSTOMER] of TIMESLOT
find stackClosed  : matrix indexed by [CUSTOMER] of TIMESLOT
find isOpenStack  : matrix indexed by [CUSTOMER,TIMESLOT] of bool
find nbStacks     : PRODUCT

minimising nbStacks

such that
    alldifferent_except(timeof,0),

    forall p1 : PRODUCT . (
        (exists p2 : PRODUCT . (
            (p1 ≠ p2) ∧
            (forall c : CUSTOMER . demand[c,p1] ≤ demand[c,p2]))
        ) ⇒ (timeof[p1] = 0)
    ),

    forall p1 : PRODUCT . (
        (forall p2 : PRODUCT . (
            (p1 ≠ p2) ⇒
            !(forall c : CUSTOMER . demand[c,p1] ≤ demand[c,p2]))
        ) ⇒ (timeof[p1] > 0)
    ),

    forall c : CUSTOMER . forall p : PRODUCT .
         (timeof[p] > 0 ∧ demand[c,p]) ⇒ stackOpened[c] ≤ timeof[p],

    forall c : CUSTOMER . forall p : PRODUCT .
        (timeof[p] > 0 ∧ demand[c,p]) ⇒ stackClosed[c] ≥ timeof[p],

    forall c : CUSTOMER . forall t : TIMESLOT .
        isOpenStack[c,t] = ((stackOpened[c] ≤ t) ∧ (t ≤ stackClosed[c])),

    forall t : TIMESLOT .
        nbStacks ≥ (sum c : CUSTOMER . isOpenStack[c,t])
```

Fig. 15. An improved model.

the propagation levels of the used constraints are similar. Some models may take far more search nodes to solve, yet can be solved faster.

In our experimental results, the fact that the *better* model runs faster than the *normal* model, as long as it needs fewer search nodes demonstrates a clear gain. Using a *partial* function variable instead of a *total* function variable not only saves from the number of search nodes, but also saves from the actual time spent during search.

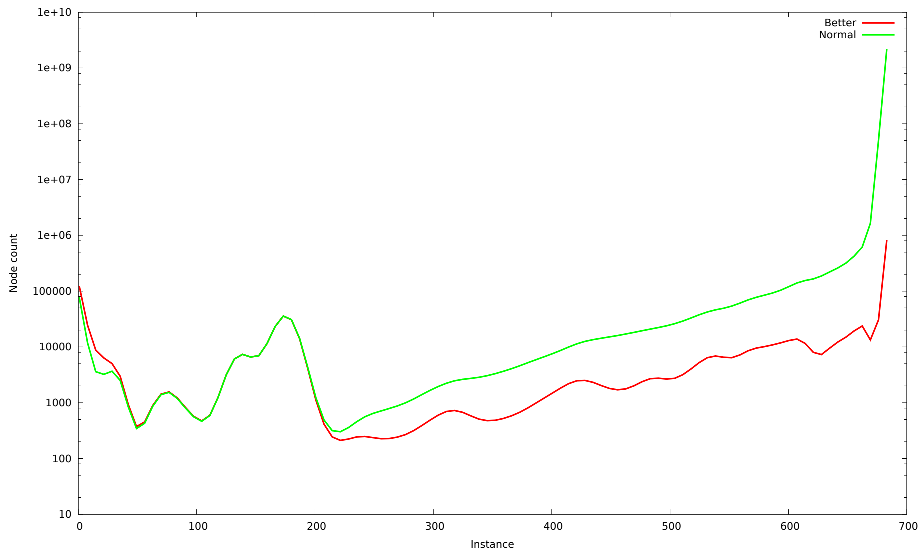

Fig. 16. Search nodes

6 Conclusion and Future work

We presented a use case for Conjure using the open stacks problem as an example. This exercise allowed us to discover a new and specific variable representation. The new refinement rules are added to the rules database; future refinements will benefit from these rules without any further work.

Conjure generates multiple models for a given problem specification. We didn't discuss how we select a good model out of the generated selection of models in this paper. This is partly because it was out of the focus of this paper and partly because we didn't develop any automated model selection techniques yet.

Now that Conjure can produce multiple valid constraint models for a given problem specification and the refinement language is mature enough to encode new transformations without modifying the internals, our aim is to study model

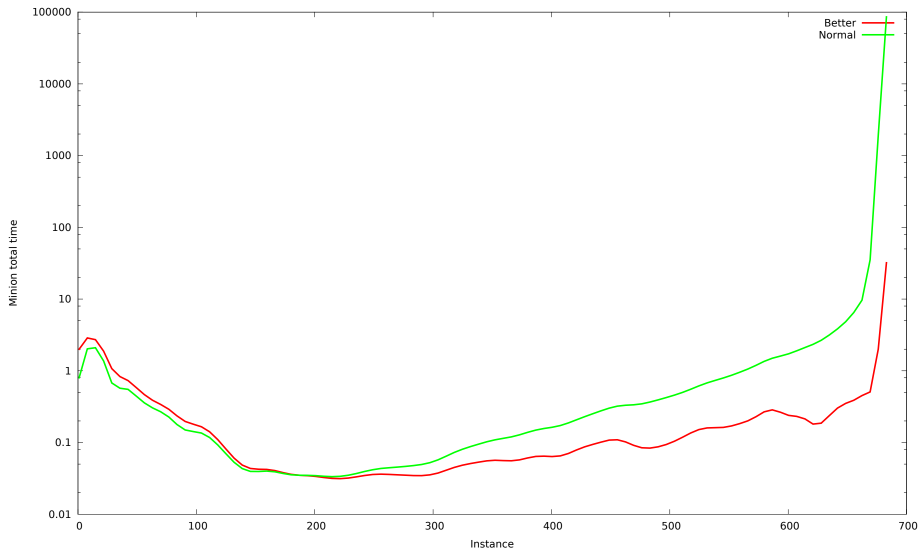

Fig. 17. Search times

selection; fully automated or assisted. We are also planning to investigate other problems to capture modelling idioms and enrich our refinement rules database.

Acknowledgements

Ozgur Akgun is supported be a Scottish Informatics and Computer Science Alliance (SICSA) prize studentship. This research is supported by UK EPSRC grant no EP/H004092/1.

References

1. B.M.W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.
2. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints 13(3)*, pages 268–306, 2008.
3. Bernadette Martínez Hernández and Alan M. Frisch. The automatic generation of redundant representations and channelling constraints. In *Trends in Constraint Programming*, chapter 8, pages 163–182. ISTE, May 2007.
4. Andrea Rendl. *Thesis: Effective Compilation of Constraint Models.* PhD thesis, University of St. Andrews, 2010.
5. Barbara M. Smith and Ian P. Gent. Constraint modelling challenge, 2005. In conjunction with The Fifth Workshop on Modelling and Solving Problems with Constraints Held at IJCAI 2005, Edinburgh, Scotland, 31 July, 2005.

Optimization Methods for the Partner Units Problem[⋆]

Markus Aschinger[1], Conrad Drescher[1], Gerhard Friedrich[2], Georg Gottlob[1], Peter Jeavons[1], Anna Ryabokon[2], Evgenij Thorstensen[1]

1 Computing Laboratory, University of Oxford
2 Institut für Angewandte Informatik, Alpen-Adria-Universität Klagenfurt

Abstract. In this work we present the Partner Units Problem as a novel challenge for optimization methods. It captures a certain type of configuration problem that frequently occurs in industry. Unfortunately, it can be shown that in the most general case an optimization version of the problem is intractable. We present and evaluate encodings of the problem in the frameworks of answer set programming, propositional satisfiability testing, constraint solving, and integer programming. We also show how to adapt these encodings to a class of problem instances that we have recently shown to be tractable.

1 Introduction

The Partner Units Problem (Pup) has recently been proposed as a new challenge in automated configuration [8]. It captures the essence of a specific type of configuration problem that frequently occurs in industry. Informally the Pup can be described as follows: Consider a set of sensors that are grouped into zones. A zone may contain many sensors, and a sensor may be attached to more than one zone. The Pup then consists of connecting the sensors and zones to control units, where each control unit can be connected to the same fixed maximum number *UnitCap* of zones and sensors.[1] Moreover, if a sensor is attached to a zone, but the sensor and the zone are assigned to different control units, then the two control units in question have to be directly connected. However, a control unit cannot be connected to more than *InterUnitCap* other control units (the partner units).
For an application scenario consider, for example, a museum where we want to keep track of the number of visitors that populate certain parts (zones) of the building. The doors leading from one zone to another are equipped with sensors. To keep track of the visitors the zones and sensors are attached to control units; the adjacency constraints on the control units ensure that communication between units can be kept simple.

⋆ Work funded by FFG FIT-IT Grant 825071 (Klagenfurt) and EPSRC Grant EP/G055114/1 (Oxford).

[1] For ease of presentation and without loss of generality we assume that *UnitCap* is the same for zones and sensors.

Let us emphasize that the PUP is not limited to this application domain: it occurs whenever sensors that are grouped into zones have to be attached to control units, and communication between units should be kept simple. Typical applications include intelligent traffic management, or surveillance and security applications. The PUP has been introduced as a novel benchmark instance at this year's answer set programming competition [2].

Figure 1 shows a PUP instance and a solution for the case *UnitCap* = *InterUnitCap* = 2. In this example six sensors (left) and six zones (right), which are completely inter-connected, are partitioned into units (shown as squares) respecting the adjacency constraints. Note that for the given parameters this is a maximal solvable instance; it is not possible to connect a new zone or sensor to any of the existing ones.

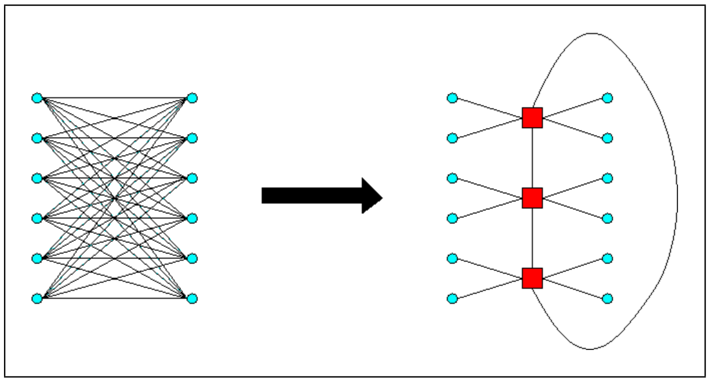

Fig. 1. Solving a $K_{6,6}$ Partner Units Instance — Partitioning Sensors and Zones into Units on a Circular Unit Layout

Very recently, we have shown that the case where *InterUnitCap* = 2 and *UnitCap* = k for some fixed k is tractable, by giving a specialized NLOGSPACE algorithm that is based on the notion of a path decomposition [1]. While this case is of some importance for our partners in industry, the general case is also interesting: Consider, for example, a grid of rooms, where every room is accessible from each neighboring room, and all the doors are fitted with a sensor. Moreover, assume there are doors to the outside on two sides of the building; the corresponding instance is shown in Figure 2, with rooms as black squares, and doors as circles. It is not hard to see that this instance is unsolvable for *InterUnitCap* = 2 and *UnitCap* = 2. However, it is easily solved for *InterUnitCap* = 4 and *UnitCap* = 2: Every room goes on a distinct unit, together with the sensors to the west and to the north; the connections between units correspond to those between rooms. Clearly this solution is optimal, in the sense of using the least possible number of units.

In this paper we present and evaluate encodings in the optimization frameworks of answer set programming, constraint satisfaction, SAT-solving, and integer programming, that can be used to solve the general version of the PUP. We also show how to adapt these encodings to the special case *InterUnitCap* $= 2$, and compare the adapted encodings against our specialized algorithm.
It is worth emphasizing that we do not take our encodings/algorithms to be the final answer on the PUP. Instead we hope that our work will spark interest in the problem across the different optimization research communities, eventually resulting in better encodings and better theoretical understanding of the problem.

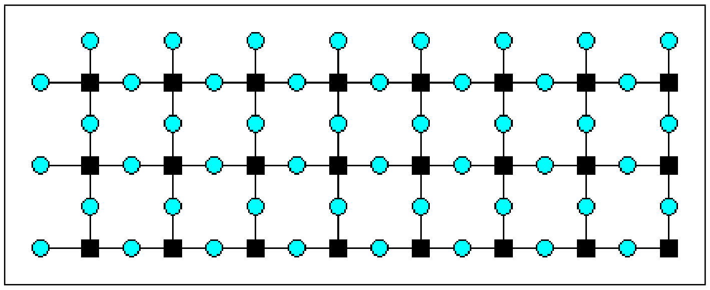

Fig. 2. A Grid-like PUP Instance

The remainder of this paper is organized as follows: In section 2 we give the basic formal definitions, and identify general properties of the PUP. Then, in section 3 we present problem models in the frameworks of answer set programming, propositional satisfiability testing, integer programming, and constraint solving; these problem models can be used for arbitrary fixed values of *InterUnitCap*. In section 4 we briefly recall our recent tractability results for the case *InterUnitCap* $= 2$, and show how the various PUP encodings presented in this paper can be adapted to this special case. Finally, in section 5 we evaluate the performance of our encodings, and in section 6 we list some directions for future research.

2 The Partner Units Problem

2.1 Formal definition

Formally, the PUP consists of partitioning the vertices of a bipartite graph $G = (V_1, V_2, E)$ into a set U of bags such that each bag

- contains at most *UnitCap* vertices from V_1 and at most *UnitCap* vertices from V_2; and
- has at most *InterUnitCap* adjacent bags, where the bags U_1 and U_2 are adjacent whenever $v_i \in U_1$ and $v_j \in U_2$ and $(v_i, v_j) \in E$.

To every solution of the Pup we can associate a solution graph. For this we associate to every bag $U_i \in \mathcal{U}$ a vertex v_{U_i}. Then the solution graph G^* has the vertex set $V_1 \cup V_2 \cup \{v_{U_i} \mid U_i \in \mathcal{U}\}$ and the set of edges $\{(v, v_{U_i}) \mid v \in U_i \wedge U_i \in \mathcal{U}\} \cup \{(v_{U_i}, v_{U_j}) \mid U_i$ and U_j are adjacent.$\}$. In the following we will refer to the subgraph of the solution graph induced by the v_{U_i} as the *unit graph*.
The following are the two most important reasoning tasks for the Pup: Decide whether there is a solution, and find an optimal solution, that is, one that uses the minimal number of control units. We are especially interested in the latter problem. For this we consider the corresponding decision problem: Is there a solution with a specified number of units? The rationale behind the optimization criterion is that (a) units are expensive, and (b) connections are cheap.

2.2 The Partner Units Problem is Intractable

By a reduction from BinPacking, it can be shown that the optimization version of the Pup is intractable when $\mathit{InterUnitCap} = 0$, and *UnitCap* is part of the input. Observe that clearly the Pup is in NP (cf. also section 3).

Theorem 1 ([1]). *Deciding whether a* Pup *instance has a solution with a given number of units is* NP*-complete, when* $\mathit{InterUnitCap} = 0$*, and* *UnitCap is part of the input.*

In practice, however, the value of *UnitCap* will typically be fixed.

2.3 Forbidden Subgraphs of the Pup

In solvable instances sensors cannot belong to arbitrarily many zones (and vice versa) [1]:

Lemma 1 (Forbidden Subgraphs of the Pup). *A* Pup *instance has no solution if it contains* $K_{1,n}$ *or* $K_{n,1}$ *as a subgraph, where* $n = ((\mathit{InterUnitCap} + 1) * \mathit{UnitCap}) + 1$.

2.4 *K*-regular Graphs

There is an interesting connection between most general solutions to the Pup and k-regular unit graphs, where a graph is k-regular if every vertex has exactly k neighbors: In k-regular unit graphs we are exploiting the *InterUnitCap* capacity for connections between units to the fullest. Hence, k-regular unit graphs are the most general solutions (if they exist). In the case where $k = 2$, there is exactly one k-regular graph, the cycle; we exploit this fact in section 4. In the case where k is odd, k-regular unit graphs only exist if there is an even number of units ("hand-shaking lemma"). Moreover, for $k > 2$ the number of distinct most general unit graphs grows rapidly: E.g. for $k = 4$ there are six distinct graphs on eight vertices, and 8037418 on sixteen vertices; for twenty vertices not all distinct graphs have been constructed [13]. It can be shown that all these solution topologies can be forced:

Observation 1 (PUP instances and k-regular graphs) *For every k-regular graph G_k there exists a* PUP *instance G with $InterUnitCap = k$ such that in every solution of G the unit graph is G_k.*

Proof. Construct the instance G as follows:
1) First connect $2 * UnitCap$ vertices (i.e. sensors and zones) to each node in G_k. Let the set of all sensors (zones) be V_1 (V_2).
2) The instance G contains all edges (v_1, v_2), where $v_1 \in V_1$ and $v_2 \in V_2$ are connected to either the same or adjacent nodes in G_k.

We show that every solution is isomorphic to G_k. We consider two cases:
- $0 \leq k \leq 1$: The result is immediate.
- $k > 1$: Let u_0 be a node in G_k with neighbors $u_j : 1 \leq j \leq k$. Denote by $V_1^{u_i}$ and $V_2^{u_i}$ the sensors and zones created in G for $u_i : 0 \leq i \leq k$. Let G'_k be an optimal solution for G. We need two observations: (1) For each $0 \leq i \leq k$ both $V_1^{u_i}$ and $V_2^{u_i}$ are on the same unit in G'_k. (2) For $0 \leq i < j \leq k$ if $V_1^{u_i} \cup V_2^{u_i}$ and $V_1^{u_j} \cup V_2^{u_j}$ are connected in G then their units are connected in G'_k.

Hence, if $InterUnitCap > 2$, and there are no restrictions on the solution topology in the application domain, then it is practically not feasible to iteratively try all most general solution topologies. The solution topology will have to be inferred instead.

2.5 Bounds on the Number of Units Required

Let us next point out that the number of units used when solving an instance $G = (V_1, V_2, E)$ is bounded from below by $lb = \lceil \frac{\max(|V_1|, |V_2|)}{UnitCap} \rceil$. Clearly it can also be bounded from above by $ub = |V_1| + |V_2|$ — we never need empty units. If $InterUnitCap = 2$ and $UnitCap > 1$ we can show that the stronger $ub = \max(|V_1|, |V_2|)$ holds for connected instances [1]. Now, if there are multiple connected components C_i in the instance with upper bounds ub_i, then we have $ub = \sum ub_i$. We conjecture that this also holds for $InterUnitCap > 2$, but have so far been unable to prove it. These bounds are exploited in the problem encodings below either for keeping the problem model small, or to limit the depth of iterative deepening search. In this approach we first try to find a solution with lb units; if that fails increase lb by one; the first solution found will be optimal. For both approaches better upper bounds are desirable.

2.6 Symmetry breaking

If we don't use iterative deepening search, then in some problem models we might obtain solutions with empty units. Here we can do symmetry breaking, by demanding that whenever unit j has a sensor or zone assigned to it, then every unit $j' < j$ also has some sensor or zone assigned to it.

We can also rule out a lot of the connections between sensors and units (or alternatively, between zones and units) immediately. Consider sensors and units: Sensor 1 must be somewhere, so it might as well be on unit 1. Sensor 2 can either be on unit 1 or on a new unit, let's say 2, and so on. Unfortunately, we cannot do this on both sensors and zones, since the edges may disallow a zone and a sensor on the same unit.

3 Encodings for the General Case

We are next going to outline encodings of the PUP where *InterUnitCap* is an arbitrary fixed constant. Due to cost considerations we are especially interested in the optimization version of the PUP: We want to minimize the number of expensive units used, but do not consider the cost for the cheap connections between them.
In particular, we show how the problem can be encoded in the frameworks of propositional satisfiability testing (SAT), integer programming (IP), and constraint solving (CSP), all of which can be considered as state-of-the-art for optimization problems [11]. In addition we will also describe an encoding in answer set programming (ASP), a currently very successful knowledge representation formalism.

3.1 Answer Set Programming

First, we show how to encode the PUP in answer set programming [9, 12] which has its roots in logic programming and deductive databases. This knowledge representation language is based on a decidable fragment of first-order logic and is extended with language constructs such as aggregation and weight constraints. Already the propositional variant allows the formulation of problems beyond the first level of the polynomial hierarchy. In case standard propositional logic is employed [2], an answer set corresponds to a minimal logical model by definition of [12].
In our encodings a solution (i.e. a configuration) is the restriction of an answer set to those literals that satisfy the defined solution schema.
To encode a PUP instance in ASP we represent the zones and sensors by the unary predicates `zone/1` and `sensor/1`. The edges between zones and sensors are represented by the binary predicate `zone2sensor/2`. The number of available units `lower` $= \left\lceil \frac{\max(|Sensors|,|Zones|)}{2} \right\rceil$, `unitCap` and `interUnitCap` are each specified by a constant. The PUP is then encoded via the following logical sentences employing the syntax described in [3]:

```
(1)  unit(1..lower).
(2)  1 { unit2zone(U,Z) : unit(U) } 1 :- zone(Z).
(3)  1 { unit2sensor(U,S) : unit(U) } 1 :- sensor(S).
(4)  :- unit(U), unitCap+1 { unit2zone(U,Z): zone(Z) }.
(5)  :- unit(U), unitCap+1 { unit2sensor(U,S): sensor(S) }.
(6)  partnerunits(U,P) :- unit2zone(U,Z), zone2sensor(Z,S),
                          unit2sensor(P,S), U!=P.
(7)  partnerunits(U,P) :- partnerunits(P,U), unit(U), unit(P).
(8)  :- unit(U), interUnitCap+1 { partnerunits(U,P): unit(P) }.
```

The first statement generates the required number of units represented as facts: `unit(1)`. `unit(2)`. ... `unit(lower)`. The second and the third

[2] All literals in rules are negation free. $\bot$, $\rightarrow$, $\wedge$, $\vee$ are used to formulate (disjunctive) rules.

clause ensure that each zone and sensor is connected to exactly one unit. The edges between units and zones (rsp. sensors) are expressed by `unit2zone/2` (rsp. `unit2sensor/2`) predicates. We use cardinality constraints [17] of the form $l\ \{L_1, \ldots, L_n\}\ u$ specifying that at least l but at most u literals of $L_1, \ldots, L_n$ must be true. So called *conditions* (expressed by the symbol ":") restrict the instantiation of variables to those values that satisfy the condition. For example, in the second rule, for any instantiation of variable `Z` a collection of ground literals `unit2zone(U, Z)` is generated where the variable `U` is instantiated to all possible values s.t. `unit(U)` is true. In this collection at least one and at most one literal must be true.
The fourth and the fifth rule guarantee that one unit controls at most *UnitCap* zones and *UnitCap* sensors. In these rules the head of the rule is empty which implies a contradiction in case the body of the rule is fulfilled. The last three rules define the connections between units and limit the number of partner units to *InterUnitCap*. Note that rules 4, 5 and 8 can be rephrased by moving the cardinality constraint on the left-hand-side of the rule and adapting the boundaries. We used the depicted encoding because it follows the Guess/Check/Optimize pattern formulated in [12]. Depending on the particular encoding runtimes may vary.
Alternatively, ASP solvers provide built-in support for optimization by restricting the set of answer sets according to an objective function. For example, for minimizing the number of units, the upper bound on the number of units used has to be provided as a constant `upper` = $\max(|Zones|, |Sensors|)$. The unit generation rule of the original program (line 1) then has to be replaced by:

```
(1’) unit(1..upper).
(2’) unitUsed(U):- unit2zone(U,Z).
(3’) unitUsed(U):- unit2sensor(U,S).
(4’) lower { unitUsed(X):unit(X) } upper.
(5’) unitUsed(X):- unit(X), unit(Y), unitUsed(Y), X<Y.
(6’) #minimize[unitUsed(X)].
```

Here, the second and the third rule express the property that a used unit always has to be non-empty. Rule `4’` states that the number of used units must be between `lower` and `upper`. Rule `5’` expresses an ordering on the units: units with smaller numbers should be used first. This statement improves the performance of the solver. The last rule expresses that the optimization criterion is the number of units used in a solution.

3.2 Propositional Satisfiability Testing

We next show how to encode the PUP as a propositional satisfiability problem. We are given sensors $[1, S]$, zones $[1, Z]$, and units $[1, U]$, as well as *UnitCap* and *InterUnitCap*.

Let su_{ij} denote that sensor i is assigned to unit j, and zu_{ij} that zone i is assigned to unit j. First of all, every sensor and zone must belong to a unit, so

$$\forall 1 \leq i \leq S \bigvee_{1 \leq j \leq U} su_{ij} \text{ and } \forall 1 \leq i \leq Z \bigvee_{1 \leq j \leq U} zu_{ij}.$$

Furthermore, every sensor and zone belongs to at most one unit, therefore we have

$$\forall 1 \leq i \leq S . \forall 1 \leq j < j' \leq U. \left(\neg su_{ij} \vee \neg su_{ij'}\right)$$

and the same for zones.
Now we need to count both the number of zones and sensors on a unit, and forbid both numbers to be above *UnitCap*. For this we use a sequential counter, similar to the one presented in [18]. Let sc_{ijk} mean that unit j has k sensors assigned (ignore the i for now). We need to say that every sensor counts as one,

$$\forall 1 \leq i \leq S . \forall 1 \leq j \leq U. \left(su_{ij} \rightarrow sc_{ij1}\right),$$

and also that we increment this number when we see something new:

$$\forall 1 \leq i < i' \leq S . \forall 1 \leq j \leq U . \forall 1 \leq k \leq \mathit{UnitCap}. \\ \left(su_{i'j} \wedge sc_{ijk} \rightarrow sc_{i'j(k+1)}\right)$$

The fact that we keep track of what we have seen (using index i) is to make sure, for example, that sc_{ij5} is only true if there are five distinct sensors on a unit. Finally, we forbid too many sensors on a unit via

$$\forall 1 \leq i \leq S . \forall 1 \leq j \leq U . \neg sc_{ij(\mathit{UnitCap}+1)}.$$

Repeat this trick for zones using zc_{ijk}.
Finally, we need to use the edges. Let sz_{ij} be given, and mean that sensor i has an edge to zone j. Also, let uu_{ij} mean that units i and j are partnered. We need to define this as

$$\forall 1 \leq i \leq S . \forall 1 \leq j \leq Z . \forall 1 \leq k < k' \leq U. \\ \left(\left(\left(su_{ik} \wedge zu_{jk'}\right) \vee \left(su_{ik'} \wedge zu_{jk}\right)\right) \wedge sz_{ij} \rightarrow uu_{kk'}\right)$$

and also, by symmetry,

$$\forall 1 \leq i < j \leq U. \left(uu_{ij} \rightarrow uu_{ji}\right).$$

Now we can count the partnered units like we did before, using pc_{ijk}, and then forbidding $pc_{ij(y+1)}$. Technically, we don't need both uu_{ij} and uu_{ji}, but having both makes the encoding simpler in the definitions above. We may skip uu_{ii} — but we may also leave them in, as the clauses forcing uu_{ij} have $i < j$, and thus uu_{ii} is never forced. Therefore,

$$\forall 1 \leq i \leq j \leq U. \left(uu_{ij} \rightarrow pc_{ij1}\right),$$

and

$$\forall 1 \leq i < i' \leq U . \forall 1 \leq j \leq U. \left(uu_{i'j} \wedge pc_{ijk} \rightarrow pc_{i'j(k+1)}\right).$$

Finally, we forbid too many partners, and we are done:

$$\forall 1 \leq i \leq j \leq U . \neg pc_{ij(\mathit{InterUnitCap}+1)}.$$

3.3 Integer Programming

We next show how the PUP can be encoded into integer programming. If *InterUnitCap* = 2 we set |Units| = max(|Sensors|, |Zones|); otherwise it is |Units| = |Sensors| + |Zones|. Then we make matrices of Boolean variables su_{ij} (and zu_{ij}, respectively) sensor s_i (zone z_i) is assigned to unit u_j. These matrices are constrained to enforce that each sensor/zone is assigned exactly one unit, and that no unit is assigned more than *UnitCap* sensors/zones:

$$\begin{array}{cccc|c}
su_{1,1} & su_{2,1} & su_{3,1} & \ldots & \sum \leq \mathit{UnitCap} \\
su_{1,2} & su_{2,2} & \ldots & \ldots & \sum \leq \mathit{UnitCap} \\
su_{1,3} & \ldots & \ldots & \ldots & \sum \leq \mathit{UnitCap} \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\hline
\sum = 1 & \sum = 1 & \ldots & \ldots &
\end{array}$$

The zone-units matrix looks identical. Next we need a Boolean variable $\mathit{UnitUsed}_i$ that indicates whether u_i is assigned any sensors/zones. This can be achieved by constraints $su_{ji} \leq \mathit{UnitUsed}_i$ and $zu_{ji} \leq \mathit{UnitUsed}_i$, for all j. Observe that in principle even for unused units $\mathit{UnitUsed}_i$ can be set to one — a possibility that will be excluded by the objective function.

For the constraints on the connections between units it is convenient to increase *InterUnitCap* by one, and stipulate that every unit is connected to itself. We then obtain a symmetric matrix of Boolean uu_{ij} variables, which can be used to indicate whether unit i is connected to unit j:

$$\begin{array}{cccc|c}
1 & uu_{1,2} & uu_{1,3} & \ldots & \sum \leq \mathit{InterUnitCap} + 1 \\
uu_{2,1} & 1 & \ldots & \ldots & \sum \leq \mathit{InterUnitCap} + 1 \\
uu_{3,1} & \ldots & 1 & \ldots & \sum \leq \mathit{InterUnitCap} + 1
\end{array}$$

In addition to enforcing that *InterUnitCap* is not exceeded, the entries in this matrix are subject to the following constraints:

- $uu_{ij} = uu_{ji}$ (symmetry); and
- $uu_{ij} \geq (su_{ki} + zu_{lj}) - 1$, for all connections (s_k, z_l) between sensors and zones — if a sensor s_k and a zone z_l are connected yet assigned different units u_i, u_j then these units are connected.

This model allows more connections between units than are actually needed, in this case mandating a post-processing step for solutions.

As a last constraint we add that the number of units used is bounded from below:

$$\lceil \frac{\max\left(|\text{Sensors}|, |\text{Zones}|\right)}{2} \rceil \leq \sum_j \mathit{UnitUsed}_j .$$

Finally, we add the objective function $\sum_j \mathit{UnitUsed}_j$, subject to minimization. As usual, first a linear relaxation with cost C is solved, and only then is the problem solved over the integers, posting the cost C as a lower bound.

3.4 Constraint Satisfaction Problem

Finally, we model the Pup as a Csp by letting sensors and zones be variables $S = \{s_1, \ldots, s_n\}$ and $Z = \{z_1, \ldots, z_m\}$. For the domains we use (a numbering of) the units $U_1, \ldots, U_n$.
We post a global cardinality constraint `gcc(`U_i`,[`$s_1, \ldots, s_n$`],`C`)` on the sensors for every U_i , where C is a variable with domain $\{0, \ldots, \mathit{UnitCap}\}$, and do likewise for the zones. These constraints ensure that each unit occurs at most *UnitCap* times in any assignment to S and Z.
Tracking connections between units via Boolean variables is done using a matrix of Boolean uu_{ij} variables as in the integer programming model, but using cardinality constraints to count the number of ones.
In addition for each connection (s, z) we post implicational constraints that exclude the value j from the domain of sensor s if z is assigned to unit i and $uu_{ij} = 0$ (and vice versa):

$$(s = U_i \wedge uu_{ij} = 0) \rightarrow z \neq U_j \text{ and } (z = U_i \wedge uu_{ij} = 0) \rightarrow s \neq U_j$$

4 A Special Case: *InterUnitCap* = 2

In this section we focus on the case where *InterUnitCap* = 2. We first briefly recall the fundamental ideas of our recent tractability results for this case; for the details the interested reader is referred to [1]. We then show how the fundamental ideas from this work can be incorporated into the Pup encodings presented above.

4.1 A Specialized Algorithm for *InterUnitCap* = 2

The basic observation in the case *InterUnitCap* = 2 is that the unit graph in a solution of a connected Pup instance is always either a path or a cycle. This holds because the number of neighbors of a unit is bounded by two. Based on this observation we have developed a non-deterministic algorithm DecPup that decides the Pup. Basically, DecPup recursively guesses the contents of the units. It turns out that this can be done in NLogSpace by exploiting the notion of a path decomposition; this non-deterministic algorithm can then be turned into a polynomial backtracking search procedure.
Let us now turn to those of the ideas we use for the DecPup algorithm that can be incorporated into the other problem models: We first observe that cyclic unit graphs are more general solution topologies than paths. Any solution that is a path can be extended to a cycle, but the converse is clearly false. Hence, for a fixed number of units used in the solution, we can assume a fixed cyclic layout of the units throughout the search. By using iterative deepening search (on the number of units used) we can find optimal solutions first.
In this context let us point out that branch-and-bound-search for optimal solutions (again on the number of units used) does not work: e.g. a $K_{6,6}$ graph does not admit solutions with more than three units.
Note also that finding optimal solutions gets more complicated if there are multiple connected components in the input graph. DecPup can then

still be used to compute optimal solutions in polynomial time — but only if there are at most logarithmically many connected components in the input graph. Part of the problem is that any two connected components may either have to be assigned to the same, or to two distinct unit graph(s). A priori it is unclear which of the two choices leads to better results. E.g. if we assume that $\mathit{UnitCap} = 2$ then two $K_{3,3}$ should be placed on one cyclic unit graph, while two $K_{6,6}$ must stand alone.
Note that with cycles for unit graphs there are two kinds of rotational symmetry: For any given solution with unit graph $U_1, \ldots, U_n, U_1$ there also are identical solutions $U_2, \ldots, U_n, U_1, U_2$, etc.; in addition, there is also $U_n, U_{n-1}, \ldots, U_1, U_n$. We can break this symmetry without additional computational cost by requiring that

- the first sensor is assigned to unit U_1; and
- the second sensor appears somewhere on the first half of the cycle.

We have prototypically implemented the DecPup algorithm in Java (for connected graphs), and will use it below in the evaluation of the other encodings for the case $\mathit{InterUnitCap} = 2$. The implementation features memoization of no-good units to avoid the rediscovery of unsolvable subproblems, and two-step forward-checking: Checking whether there is enough space for the open neighbours of the current unit on the current plus the next unit (step one), and doing the same for the open neighbours of the open neighbours (step two).

4.2 Adapting the Encodings to *InterUnitCap* = 2

To some extent the ideas presented above can be incorporated into the other problem models: If we use iterative deepening search, then we can assume a fixed cyclic layout of the units for each depth. Then, the connections between units are given, something that greatly simplifies the problem models. It also allows us to use symmetry breaking as defined in section 4.1 above. For example, in the constraint model we can drop the Boolean matrix that tracks the connections between units, and simplify the implicational constraints for a connection (s, z) to

$$s = U_i \rightarrow z \in \{U_{i-1}, U_i, U_{i+1}\} \text{ and } z = U_i \rightarrow s \in \{U_{i-1}, U_i, U_{i+1}\}.$$

The adaptations for Asp and Sat are similar [1].
If we are not doing iterative deepening search, that is, the maximum available number of units in the model is given by the upper bound, then this does not work, as it is not clear where to close the cycle. Especially for the integer programming model this constitutes a challenge: If we use iterative deepening we lose the objective function.
To guide the search, we can, by a simple greedy algorithm, compute an ordering of the variables that ensures that each sensor (or zone) has some predecessor that has already been assigned to a unit; we assume that an arbitrary sensor (or zone) is fixed initially. If variables are assigned in this order then the number of possible unit choices per zone (or sensor) is bounded by three throughout the search, instead of *NoOfUnits*. However, to the best of our knowledge neither integer programming tools nor Asp- or Sat-solvers usually provide this level of control over variable ordering to the user.

5 Evaluation

We have evaluated our encodings on a set of benchmark instances that we received from our partners in industry. All experiments were conducted on a 3 GHz dual core machine with 4 GB RAM running Fedora Linux, release 13 (Goddard). In general in our experiments we have imposed a ten minute time limit for finding solutions.
For the evaluation of the different encodings of the Pup we use the Sat-solver MiniSat v2.0 [14], the constraint logic programming language ECL^{i}PS^e-Prolog v6.0 [7], and Clingo v3.0 [3] from the Potsdam Answer Set Solving Collection (Potassco). For evaluating the integer programming model we have used Cbc v2.6.2 in combination with Clp v1.13.2 from the COIN-OR project [4], and IBM's Cplex v12.1 [5].
In the Asp, Sat and Csp models, as well as in DecPup, we use iterative deepening search for finding optimal solutions, as this has proven to be the most efficient. We did not try this in the integer programming model, as we would lose the objective function in doing so.
The reader is advised to digest the results presented below with caution: We are using both the Sat and the integer programming solvers out of the box, whereas for the Csp model we employ the variable ordering heuristics outlined in the previous section. Moreover, if *InterUnitCap* > 2, for the Asp model we employ the following advanced feature: a portfolio solver Claspfolio, which is a part of Potassco [3], comes with a machine learning algorithm (support vector machine) that has been trained on a large set of Asp programs. Claspfolio analyzes a new Asp program (in our case the Pup program), and configures Clingo to run with options that have already proved successful on similar programs. It is likely that such machine learning techniques could also be developed and fruitfully applied in the other frameworks.

5.1 Experimental Results

InterUnitCap $= 2$ All instances are based on rectangular floor plans, and all instance graphs are connected. In all instances there is one zone per room, and by default there are sensors on all doors. Only the grid-* and tri-* instances feature external doors. For an illustration see Figure 2, which shows a rectangular 8×3 floor plan with external doors on two sides of the building.
Apart from that, the instances are structured as follows:
- dbl-* consist of two rows of rooms with all interior doors equipped with a sensor.
- dblv-* are the same, only that there are additional zones that cover the columns.
- tri-* are grids with only some of the doors equipped with sensors. There are additional zones that cover multiple rooms.
- grid-* are not full grids, but some doors are missing, and there are no rooms (zones) without doors.

The runtimes we obtained for the various problem encodings described above are shown in seconds in Table 1 (a "*" indicates a timeout). The Cost column contains the number of units in an optimal solution; a slash "/" in that column indicates that no solution exists.

Table 1. Structured Problems with *InterUnitCap* = *UnitCap* = 2

Name	$\|S\|$	$\|Z\|$	Edges	Cost	Csp	Sat	DecPup	Asp	Cbc	Cplex
dbl-20	28	20	56	14	0.02	0.48	0.01	0.16	14.12	1.53
dbl-40	58	40	116	29	0.28	2.36	0.05	3.93	224.14	13.58
dbl-60	88	60	176	44	0.42	29.74	0.08	*	*	213.58
dbl-80	118	80	236	59	1.14	*	0.16	*	*	522.50
dbl-100	148	100	296	74	1.89	*	0.41	*	*	*
dbl-120	178	120	356	89	3.21	*	0.39	*	*	*
dbl-140	208	140	416	104	5.01	*	0.59	*	*	*
dbl-160	238	160	476	119	13.94	*	0.71	*	*	*
dbl-180	268	180	536	134	20.07	*	0.87	*	*	*
dbl-200	298	200	596	149	14.4	*	1.08	*	*	*
dblv-30	28	30	92	15	0.09	0.42	65.49	0.26	37.18	2.93
dblv-60	58	60	192	30	0.26	3.15	*	1.94	*	*
dblv-90	88	90	292	45	0.82	12.54	*	27.35	*	*
dblv-120	118	120	392	60	1.85	41.65	*	13.92	*	*
dblv-150	148	150	492	75	3.48	20.97	*	29.54	*	*
dblv-180	178	180	592	90	6.20	44.28	*	54.50	*	*
tri-30	40	30	78	20	1.07	0.79	0.50	0.41	45.17	78.75
tri-32	40	32	85	20	0.64	0.74	*	0.26	55.20	4.66
tri-34	40	34	93	/	21.10	22.77	*	0.89	74.78	5.06
tri-60	79	60	156	40	158.49	315.42	114.08	4.40	*	108.01
tri-64	79	64	170	/	*	379.36	*	43.88	*	76.26
grid-90	50	68	97	34	0.04	4.51	0.03	1.53	*	21.19
grid-91	50	63	97	32	0.10	*	*	0.92	*	16.60
grid-92	50	65	97	33	0.49	*	*	0.87	*	17.40
grid-93	50	58	97	29	0.13	2.68	*	1.75	*	13.41
grid-94	50	66	97	33	0.04	3.66	*	1.61	*	*
grid-95	50	60	97	30	0.02	3.90	0.48	0.97	*	18.34
grid-96	50	62	97	31	0.07	3.30	*	0.87	*	13.62
grid-97	50	64	97	32	0.02	3.67	*	0.86	*	17.90
grid-98	50	59	97	30	0.03	*	*	1.19	*	12.30
grid-99	50	65	97	33	0.03	*	202.48	1.16	*	20.35

InterUnitCap > 2 For the general case we have also tested our encodings on a set of benchmark instances where *InterUnitCap* $= 4$ that we obtained from our partners in industry:

- tri-* are exactly as before, only with *InterUnitCap* $= 4$.
- grid-* are as before, only that a bigger number of doors exists.

5.2 Analysis

Any conclusions drawn from our experimental results have to be qualified by the remark that, of course, in every solution framework there are many different problem models, and there is no guarantee that our problem models are the best ones possible.

Table 2. Structured Problems with *InterUnitCap* = 4 and *UnitCap* = 2

Name	$\|S\|$	$\|Z\|$	Edges	Cost	Csp	Sat	Asp	Cbc	Cplex
tri-30	40	30	78	20	0.12	2.40	0.40	182.91	24.79
tri-32	40	32	85	20	0.14	1.91	0.66	270.27	20.84
tri-34	40	34	93	20	*	1.98	0.60	331.29	*
tri-60	79	60	156	40	0.52	*	11.07	*	*
tri-64	79	64	170	40	*	*	7.61	*	*
tri-90	118	90	234	59	1.50	401.44	332.34	*	*
tri-120	157	120	312	79	3.37	*	*	*	*
grid-1	100	79	194	50	*	78.19	31.45	*	*
grid-2	100	77	194	50	*	90.89	18.91	*	*
grid-3	100	78	194	50	*	88.87	25.72	*	*
grid-4	100	80	194	50	*	95.12	24.66	*	*
grid-5	100	76	194	50	*	454.42	48.88	*	*
grid-6	100	78	194	50	*	204.85	9.15	*	*
grid-7	100	79	194	50	*	112.36	12.89	*	*
grid-8	100	78	194	50	*	*	11.89	*	*
grid-9	100	76	194	50	*	91.62	19.71	*	*
grid-10	100	80	194	50	*	545.16	13.54	*	*

Let us begin our analysis of the results by highlighting a peculiarity of the Pup: While it is possible to construct instances that require more than the minimum number of units, it is not straight-forward to do so, and such instances also appear to be rare in practice: In our experiments in no solution are there more units than the bare minimum required. It is clear that iterative deepening search thrives on this fact, whereas the integer programming model suffers.

InterUnitCap = 2 The combination of assuming a fixed cyclic unit graph together with iterative deepening search resulted in drastic speed-ups for the Asp, Sat, and Csp solvers. Symmetry breaking did not have much effect — except on the unsolvable instances.
The Asp and the Sat encoding show broadly similar behavior: Both Clingo and MiniSat use variations of the DPLL-procedure [6] for reasoning. Oddly, they even both get faster at some point as problem size increases on the dblv-* instances. However, Clingo does significantly better on the grid-like instances. Interestingly, machine learning did not help for the Asp encoding specialized to *InterUnitCap* = 2; hence the results shown were obtained using both solvers out of the box.
For the Csp encoding the variable ordering is the key to the good results: Without the variable ordering the Csp model performs quite poorly. The absence of a similar variable selection mechanism from both Asp and Sat in our experiments might explain the surprising superiority of Csp on most benchmarks.
The inconsistent results for DecPup are particularly striking. On the one hand, DecPup performs excellently on the dbl-* instances. But in general, it disappoints. Possibly this might be due to the following: DecPup

has a “local” perspective on the problem, that is, it only can see the current and past units; the subsequent units are only created at runtime. In all the other encodings all units are present from the beginning, something which, in one way or another, facilitates propagating the current variable assignment to other units.

The Ip encoding is not yet fully competitive. It particularly struggles with the dblv-* instances. In general, the commercial Cplex is at least one order of magnitude faster than the open source Cbc.

It is also interesting to compare the dbl-* with the dblv-* instances, as the latter are obtained from the former by adding constraints. Both Clingo and MiniSat thrive on the additional constraints, contrary to ECL^{i}PS^e, Cbc, Cplex and DecPup.

InterUnitCap > 2 In this setting, for finding solutions the symmetry breaking methods from section 2.6 did increase computation time for the Csp, the Sat, and the Ip model. However, symmetry breaking again does help when proving an instance unsatisfiable. The results in Table 2 were obtained without symmetry breaking.

If Claspfolio’s machine learning database is not used to configure options of Clingo, then the two DPLL-based programs again perform quite similar, with Clingo slightly having the edge (results not shown). With machine learning Clingo clearly is the winner, with the main benefits stemming from the following: Use the VSIDS heuristics [15] instead of the BerkMin heuristics [10], and exploit local restarts [16]. Note that MiniSat also uses the VSIDS heuristics.

Interestingly, the Csp-encoding now disappoints. Given that the same variable ordering is used, this may have to be attributed to insufficient propagation when tracking the connections between units.

Again our Ip encoding is not on par yet. But for this encoding comparing the instances tri-30,32,34 in Tables 1 and 2 is particularly instructive: This is basically the same model in both settings, only that in the latter case there are more variables due to the higher upper bound on the number of required units.

6 Future Work

There is still significant work to be done on the Pup: Almost all interesting complexity questions are still open, and a thorough investigation of these questions should eventually lead to better algorithms and encodings for the Pup. It should also be possible to prove better upper bounds, in particular ones that depend on *UnitCap*; especially the integer programming model would benefit from this. It would be interesting to see what the variable ordering heuristics can do for Sat and Asp. More generally, the major challenge is to find stronger problem models in the various frameworks and to improve the implementation of DecPup, the only algorithm guaranteed to run in polynomial time.

References

1. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., , Ryabokon, A., Thorstensen, E.: Tackling the Partner Units Problem. Tech. Rep. RR-10-28, Computing Laboratory, University of Oxford (2010), available from the authors
2. Third International Answer Set Programming Competition 2011. https://www.mat.unical.it/aspcomp2011/ (2011)
3. The Potsdam Answer Set Solving Collection. http://potassco.sourceforge.net/
4. COIN-OR CLP/CBC IP solver. http://www.coin-or.org/
5. IBM ILOG CPLEX IP solver. http://www.ibm.com/
6. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3) (1960)
7. ECL^{i}PS^e-Prolog. http://eclipseclp.org/
8. Falkner, A., Haselböck, A., Schenner, G.: Modeling Technical Product Configuration Problems. In: Proceedings of the Configuration Workshop at ECAI'10 (2010)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP'88 (1988)
10. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proceedings of DATE'02 (2002)
11. Hooker, J.N.: Integrated Methods for Optimization. Springer, New York (2006)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3) (2006)
13. Meringer, M.: Regular Graphs Page. http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html
14. Minisat SAT solver. http://www.minisat.se
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of DAC'01 (2001)
16. Ryvchin, V., Strichman, O.: Local restarts. In: Proceedings of SAT'08 (2008)
17. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2) (2002)
18. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Proceedings of CP 2005. Springer (2005)

Equivalence in QCSP(QBF)

Vincent Barichard and Igor Stéphan

LERIA University of Angers, France
email: {barichard, stephan}@info.univ-angers.fr

Abstract. The QBF validity problem with any propositional formula may be seen as a QCSP. In this frame, we introduce a new constraint: the equivalence constraint and prove that it is stronger than equality constraint. In order to design this equivalence constraint, we introduce a new sequent calculus for QBF which is based on an equivalence relation over subformulae. This sequent calculus is proven to be sound and complete w.r.t. the semantics of the QBF. Based on this system, we define our equivalence constraint in such a way that QCSP(QBF) may be seen as a CSP(QBF) with a quantified search algorithm. We report some experiments in a generic constraint development environment.

1 Introduction

The *Quantified Boolean Formula* (QBF) formalism has been initially introduced in [19] and more deeply study in [27]. At the very beginning this formalism which describes a "hierarchy" of languages was a theoretical tool to describe complexity classes beyond the NP class. But after [8] it also became a field of applied artificial intelligence with many implementations of decision procedures and it has been used in many fields of computer science (for example in planning [22], as a target language for implementation of other logical formalisms [5] or in formal verification [4]). This formalism is a restriction of second-order logic but is better seen as an extension of propositional logic where propositional symbols are quantified: a QBF is constituted of a string $q_1x_1 \dots q_nx_n$, the binder (with $x_1, \dots, x_n$ distinct propositional symbols and $q_1 \dots q_n$ quantifiers) and a propositional formula, the matrix. As usual, $\exists$ stands for the existential quantifier and $\forall$ stands for the universal quantifier. For example, $\forall a \forall b \exists c(\neg(a \leftrightarrow c) \rightarrow (b \leftrightarrow c))$ is a QBF ($\neg$ stands for negation, $\rightarrow$ stands for implication, $\leftrightarrow$ stands for bi-implication and a, b and c denote propositional symbols). Propositional connectors keep their usual propositional semantics and quantifier semantics is inspired by first-order logic (x denotes a propositional symbol, ϕ a QBF, $\vee$ stands for disjunction, $\wedge$ stands for conjunction, the propositional constant $\top$ stands for what is always true and the propositional constant $\perp$ stands for what is always false, $[\ \leftarrow\ ]$ stands for substitution):

$$\exists x\phi = ([x \leftarrow \perp](\phi) \vee [x \leftarrow \top](\phi))$$

and

$$\forall x\phi = ([x \leftarrow \perp](\phi) \wedge [x \leftarrow \top](\phi)).$$

A QBF ϕ is valid if $\phi \equiv \top$ (where $\equiv$ stands for logical equivalence). For example, the QBF $\exists y \exists x \forall z((x \vee y) \leftrightarrow z)$ is not valid but the QBF $\forall z \exists y \exists x((x \vee y) \leftrightarrow z)$ is valid. The satisfiability problem (SAT) of propositional logic is nothing more than the validity problem for QBF constituted of a propositional formula embedded in existential quantifiers associated with their propositional symbols. Hence, most of the more recent decision procedures for the validity problem of QBF (see [15]) are based on the (propositional version of the) search-based algorithm of Davis, Logemann and Loveland [9] (DLL) for SAT which is a direct consequence of the semantics of the existential quantifier. (Some other decision procedures are based on the resolution principal such that the Q-resolution [17] or Quantor [6] ; some others like QSAT [21] or QMRES [20] are based on quantifier elimination "à la" Fourier-Motzkin; some others like skizzo [3] are based on Skolemization).

Conjunctive normal form (CNF) is a privileged fragment for SAT and search-based algorithms because of local unsatisfiability detection. Since most of the DLL-based decision procedures for QBF are extension of SAT procedures [22, 13], there are also based on QBF in CNF. But this fragment is not adapted to QBF because it does not reflect the duality of quantifiers which is a direct consequence of the duality of conjunction and disjunction. Hence, a mix of DNF (disjunctive normal form) and CNF [28, 23], negation normal form (NNF) [2, 18, 10], or even larger fragments[1] [16] are used in modern decision procedures for QBF validity problem.

If any QBF are considered, by introduction of new propositional symbols [7](or literals [26]), QBF validity problem may be seen as a *Quantified Constraint Satisfaction Problem* (QCSP) [12]. For example, $\forall a \forall b \exists c(((a \to c) \to \neg(c \to a)) \to \neg((b \to c) \to \neg(c \to b)))$ is valid[2] if and only if the following decomposed QBF is valid[3]

$$\begin{array}{l}\forall a \forall b \exists c \exists o_0 \exists o_1 \exists o_2 \exists o_3 \exists o_4 \exists o_5 \\ \quad ((o_2 \to \overline{o_5}) \leftrightarrow \top) \wedge ((o_0 \to \overline{o_1}) \leftrightarrow o_2) \wedge ((a \to c) \leftrightarrow o_0) \wedge ((c \to a) \leftrightarrow o_1) \\ \wedge ((o_3 \to \overline{o_4}) \leftrightarrow o_5) \wedge ((b \to c) \leftrightarrow o_3) \wedge ((c \to b) \leftrightarrow o_4)\end{array}$$

only if every QBF (the constraints, denoted c_i) in the following set are valid

$$\left\{\begin{array}{ll} c_1 : \exists o_2 \exists o_5((o_2 \to \overline{o_5}) \leftrightarrow \top), & c_2 : \exists o_0 \exists o_1 \exists o_2((o_0 \to \overline{o_1}) \leftrightarrow o_2), \\ c_3 : \forall a \exists c \exists o_0((a \to c) \leftrightarrow o_0), & c_4 : \forall a \exists c \exists o_1((c \to a) \leftrightarrow o_1), \\ c_5 : \exists o_3 \exists o_4 \exists o_5((o_3 \to \overline{o_4}) \leftrightarrow o_5), & c_6 : \forall b \exists c \exists o_3((b \to c) \leftrightarrow o_3), \\ c_7 : \forall b \exists c \exists o_4((c \to b) \leftrightarrow o_4) & \end{array}\right\}$$

Fig.1 shows search trees where nodes (but the root) are labelled with substitutions on $\{\top, \bot\}$ and arcs are labelled with propagations; in both trees, leafs

[1] In [16], the fragment is based on $\{\neg, \wedge, \vee\}$.

[2] Since $(((a \to c) \to \neg(c \to a)) \to \neg((b \to c) \to \neg(c \to b))) \equiv ((a \leftrightarrow c) \vee (b \leftrightarrow c))$, $\forall a \forall b \exists c(((a \to c) \to \neg(c \to a)) \to \neg((b \to c) \to \neg(c \to b)))$ is valid if and only if $\forall a \forall b \exists c((a \leftrightarrow c) \vee (b \leftrightarrow c))$ is also valid.

[3] complementary operator $\overline{(.)}$ is such that if x is a propositionnal symbol then $\overline{x} = \neg x$, in particular $\overline{\overline{x}} = x$; x and $\overline{x}$ are litterals.

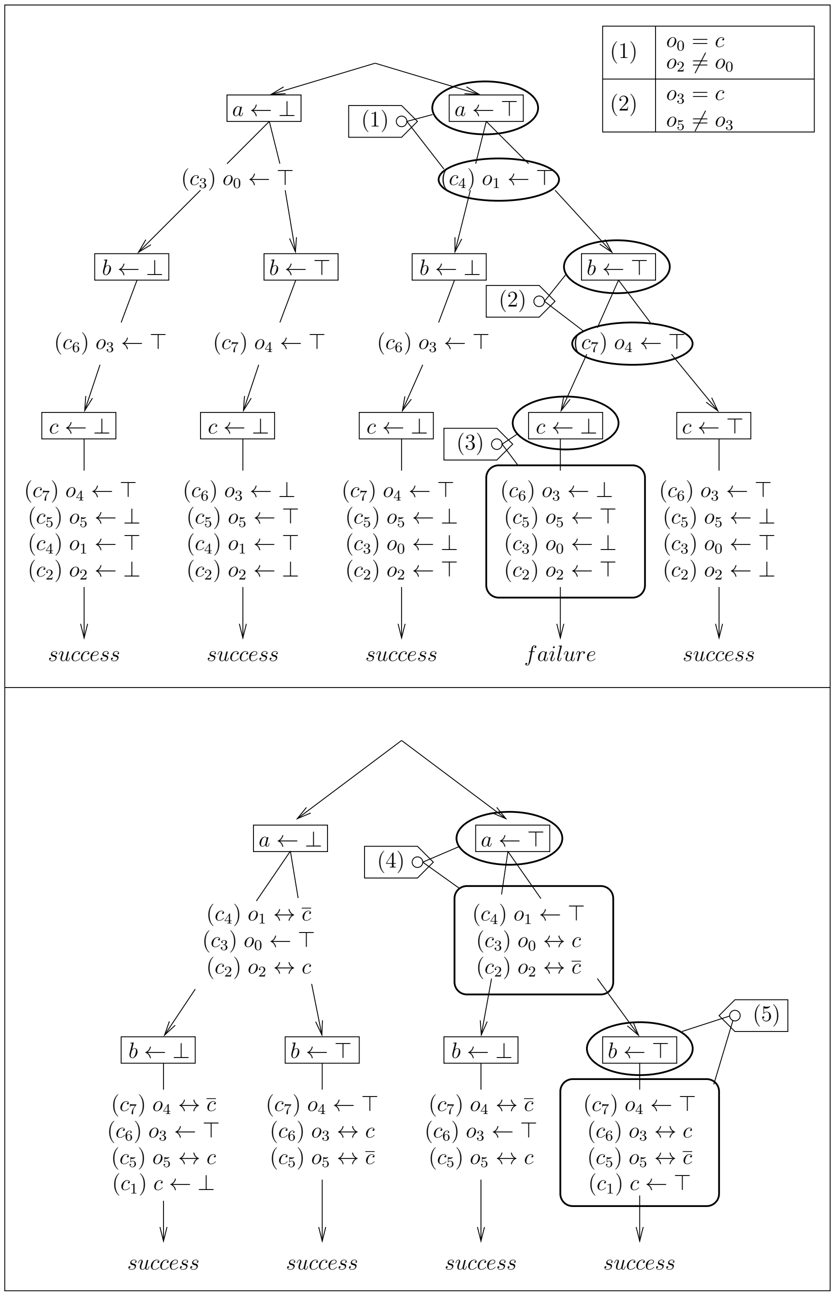
(1) $o_0 = c$ $o_2 \neq o_0$
(2) $o_3 = c$ $o_5 \neq o_3$
$a \leftarrow \bot$
$a \leftarrow \top$
$(c_3)\ o_0 \leftarrow \top$
$(c_4)\ o_1 \leftarrow \top$
$b \leftarrow \bot$
$b \leftarrow \top$
$(c_6)\ o_3 \leftarrow \top$
$(c_7)\ o_4 \leftarrow \top$
$c \leftarrow \bot$
$c \leftarrow \top$
(3)
$(c_7)\ o_4 \leftarrow \top$ $(c_5)\ o_5 \leftarrow \bot$ $(c_4)\ o_1 \leftarrow \top$ $(c_2)\ o_2 \leftarrow \bot$
$(c_6)\ o_3 \leftarrow \bot$ $(c_5)\ o_5 \leftarrow \top$ $(c_4)\ o_1 \leftarrow \top$ $(c_2)\ o_2 \leftarrow \bot$
$(c_7)\ o_4 \leftarrow \top$ $(c_5)\ o_5 \leftarrow \bot$ $(c_3)\ o_0 \leftarrow \bot$ $(c_2)\ o_2 \leftarrow \top$
$(c_6)\ o_3 \leftarrow \bot$ $(c_5)\ o_5 \leftarrow \top$ $(c_3)\ o_0 \leftarrow \bot$ $(c_2)\ o_2 \leftarrow \top$
$(c_6)\ o_3 \leftarrow \top$ $(c_5)\ o_5 \leftarrow \bot$ $(c_3)\ o_0 \leftarrow \top$ $(c_2)\ o_2 \leftarrow \bot$
success
failure
(4)
(5)
$(c_4)\ o_1 \leftrightarrow \overline{c}$ $(c_3)\ o_0 \leftarrow \top$ $(c_2)\ o_2 \leftrightarrow c$
$(c_4)\ o_1 \leftarrow \top$ $(c_3)\ o_0 \leftrightarrow c$ $(c_2)\ o_2 \leftrightarrow \overline{c}$
$(c_7)\ o_4 \leftrightarrow \overline{c}$ $(c_6)\ o_3 \leftarrow \top$ $(c_5)\ o_5 \leftrightarrow c$ $(c_1)\ c \leftarrow \bot$
$(c_7)\ o_4 \leftarrow \top$ $(c_6)\ o_3 \leftrightarrow c$ $(c_5)\ o_5 \leftrightarrow \overline{c}$
$(c_7)\ o_4 \leftrightarrow \overline{c}$ $(c_6)\ o_3 \leftarrow \top$ $(c_5)\ o_5 \leftrightarrow c$
$(c_7)\ o_4 \leftarrow \top$ $(c_6)\ o_3 \leftrightarrow c$ $(c_5)\ o_5 \leftrightarrow \overline{c}$ $(c_1)\ c \leftarrow \top$

Fig. 1. Search tree without and with equivalence constraint

are labelled with *success* or *failure*. We develop in a first time a branch of the search tree for the upper part of the figure. If a is substituted by $\top$ then, by propagation, the Boolean domain of o_1 is restricted to $\top$ by constraint c_4, and according to CSP definitions [1], an equality constraint between o_0 and c by constraint c_3 and an inequality constraint between o_2 and o_0 by constraint c_2 may be added to the set of constraints; this choice on a and its consequences are labelled by (1) in the tree and in the table. According to the semantics, a failure backtracks on the last existential propositional symbol while a success backtracks on the last universal propositional symbol (if there is no more, it is a global success). Continuing the example, if b is substituted also by $\top$ then the Boolean domain of o_4 is restricted to $\top$ by constraint c_7, and an equality constraint between o_3 and c by constraint c_6 and an inequality constraint between o_5 and o_3 by constraint c_5 may be added to the set of constraints; this is labelled by (2). At last, if c is substituted by $\bot$ then the Boolean domain of o_3 is restricted to $\bot$ by constraint c_6, the Boolean domain of o_5 is restricted to $\top$ by constraint c_5, the Boolean domain of o_0 is restricted to $\bot$ by constraint c_3 and the Boolean domain of o_2 is restricted to $\top$ by constraint c_2: it is consistent with the equality and inequality constraints but inconsistent with constraint c_1 (from which the failure); this labbeled by (3). Fig 1 shows also a search tree but for a propagation mechanism which takes into account equivalences. Coming back to a substituted by $\top$, according to Boolean properties, Boolean domain of o_1 is still restricted to $\top$ by constraint c_4, but two equivalences $(c \leftrightarrow o_0)$ and $(\overline{o_0} \leftrightarrow o_2)$ may be considered; this is labelled by (4). Continuing the example, if b is substituted also by $\top$ then the Boolean domain of o_4 is still restricted to $\top$ by constraint c_7, and two equivalences $(c \leftrightarrow o_3)$ and $(\overline{o_3} \leftrightarrow o_5)$ may be considered. Although Boolean domains of propositional symbols o_2 and o_5 of $c_1 : ((o_2 \rightarrow \overline{o_5}) \leftrightarrow \top)$ have not changed, by the equivalences $(c \leftrightarrow o_0)$, $(\overline{o_0} \leftrightarrow o_2)$, $(c \leftrightarrow o_3)$ and $(\overline{o_3} \leftrightarrow o_5)$, c restricted to $\top$ is propagated (since c_1 is by equivalence $((\overline{c} \rightarrow c) \leftrightarrow \top)$); this is labelled by (5). In the search tree for propagation without equivalence, the number of explored nodes is 11 since it is only 6 in the search tree for propagation with equivalence. Moreover, the failure has been avoided. This example proves that propagation which handles equivalence constraint is stronger than propagation without equivalence.

One way to handle equivalence, according to order induced by the binder, is to replace a propositional symbol by its equivalent by reference [26] or by rewriting their constraints: in both cases we are out of the classical frame of CSP [1]. In order to design a new equivalence constraint, we introduce a new sequent calculus for QBF which is based on an equivalence relation over subformulae. This sequent calculus is proved to be sound and complete w.r.t. the semantics of the QBF. Based on this system, we define our equivalence constraint in such a way that QCSP(QBF) may be seen as a CSP(QBF) with a quantified search algorithm. We report some experiments in the generic constraint development environment Gecode [24].

2 Preliminaries

QBF. The set of propositional symbols is denoted $\mathcal{PS}$. The set of propositional formulae is denoted **PROP**. A literal is a propositional symbol or its negation. The complementary of a propositional formula F, denoted $\overline{F}$, is G if $F = \neg G$ and $\neg F$ otherwise. The set of the sub-formulae and their complementaries of a propositional formula F, including $\top$ and $\overline{\top}$, is denoted $sub(F)$. A substitution is a function from the set of propositional symbols to **PROP**. We define the substitution of a propositional symbol x by F in G, denoted $[x \leftarrow F](G)$, as the formula obtained from G by replacing all the instances of the propositional symbol x by the formula F. An empty binder is denoted ε. A binder Q induces an order relation on the propositional symbols which is denoted $<_Q$. A model of a QBF QM is a sequence $f_1, \dots, f_n$ of propositional formulae such that $[e_1 \leftarrow f_1] \dots [e_n \leftarrow f_n](M)$ is a tautology and each formula f_i is only built over the universally quantified propositional symbols preceding, according to $<_Q$, the existentially quantified propositional symbol e_i of the binder Q. To the usual validity-preserving equivalence denoted $\equiv$, we add a model-preserving equivalence [26] denoted $\cong$ and defined as follows. Let F and G be two QBF with same binders. $F \cong G$ if F and G have the same models. For example, $\forall z \exists y \exists x((x \vee y) \leftrightarrow z) \equiv \top$ but $\forall z \exists y \exists x((x \vee y) \leftrightarrow z) \ncong \top$.

QCSP. A QCSP is a tuple $\langle \mathbf{V}, \mathrm{rank}, \mathrm{quant}, \mathbf{D}, \mathbf{C} \rangle$ where $\mathbf{V}$ is a set of n variables, rank is a bijection from $\mathbf{V}$ to $[1..n]$, quant is a mapping from $\mathbf{V}$ to $\{\exists, \forall\}$, $\mathbf{D}$ is a mapping from $\mathbf{V}$ to a set of domains $\{D(v_1), \dots, D(v_n)\}$ where, for each $v_i \in \mathbf{V}$, $D(v_i)$ is the finite domain of its possible values, $\mathbf{C}$ is a set of constraints. If $v_{i_1}, \dots, v_{i_k}$ are the variables of a constraint c_i then the relation associated to c_i is a subset of the Cartesian product $D(v_{i_1}) \times \dots \times D(v_{i_k})$.

3 Sequent calculus for quantified Boolean formulae

We present our new sequent calculus for QBF as an extension of the formula relation framework, proof-theoretical justification of the Stålmarck's method [25].

3.1 Formula Relation

A (propositional) formula relation [25] R for a formula F is an equivalence relation with domain $sub(F)$ with the constraint that, for two elements $A, B \in sub(F)$, if $R(A, B)$ then $R(\overline{A}, \overline{B})$. A formula relation class R which contains A is denoted $[A]_R$ or, when it is clear from the context, $[A]$. The meaning is that all the elements of the same equivalence class have the same truth value. We extend this formalism to a QBF QM by adding a binder to the partition P of $sub(M)$ and writing the formula relation (Q, P).

The smallest formula relation is for a QBF QM the identity relation Id_{QM}. Associated to the Q, it simply places each element of $sub(F)$ in its own equivalence class. Formula relation $R([A]_R = [B]_R)$ (simply denoted $R([A] = [B])$ in what follows) for a QBF QM is the least formula relation for QM containing R and such that $([A]_R = [B]_R)$. Formula relation $Id_{QM}([M] = [\top])$ will later be the starting point of our deduction trees of our sequent calculus. In what follows, only one of the two symmetric equivalence classes $[A]$ and $[\overline{A}]$ will be shown.

3.2 Sequent Calculus for QBF based on Formula Relation

We present our sequent calculus S_{QBF} for QBF as a set of rules $\frac{conclusion}{premise}$ which are applied on formula relations to form a deduction tree whose root is the formula relation $Id_{QM}([M] = [\top])$. But before the description of our system we define a set of formulae relation which corresponds to formula relations from which no deduction is possible anymore.

Definition 1 (Explicitly contradictory). *A formula relation is* explicitly contradictory *if one of the following conditions is verified:*

1. *there exists a formula F in the formula relation such that $[F] = [\overline{F}]$;*
2. *there exists a class which contains at least two universally quantified propositional symbols of the binder;*
3. *there exists a class which contains a universally quantified propositional symbol u of the binder and an existentially quantified propositional symbol e such that $e < u$.*

Condition 1 asserts intuitively that a formula and its complementary can not have the same truth value. Condition 2 asserts that two universally quantified symbols are never functionally linked. Condition 3 corresponds in the unification of first-order terms to the occurrence test. A formula relation might be not explicitly contradictory and containing classes which are actually contradictory.

We only present the rules of the S_{QBF} system for the fragment $\{\to, \top\}$. To simplify, in the description of the premise and the conclusion of a rule, only the binder and the relevant classes are written (invariant classes are not written, there is always implicitly the $[\top]$ class).

Definition 2 (S_{QBF} system for $\{\to, \top\}$). *The S_{QBF} system is constituted of two elimination rules for the existential quantifier:*

$$\exists\top : \frac{(Q,[x]=[\top])}{(\exists x Q,[x])} \quad \exists\bot : \frac{(Q,[\overline{x}]=[\top])}{(\exists x Q,[x])}$$

of one elimination rule for the universal quantifier:

$$\forall : \frac{(Q,[x]=[\top]) \quad (Q,[\overline{x}]=[\top])}{(\forall x Q,[x])}$$

and nine class fusion rules:

$$1 : \frac{(Q,[\overline{F_X}]=[\top],[\overline{F_Y}]=[\top])}{(Q,[(F_X \to F_Y)]=[\overline{F_Y}])} \quad 2 : \frac{(Q,[F_X]=[\top],[F_Y]=[\top])}{(Q,[(F_X \to F_Y)]=[F_X])} \quad 3 : \frac{(Q,[F_X]=[\top],[\overline{F_Y}]=[\top])}{(Q,[\overline{(F_X \to F_Y)}]=[\top])}$$

$$4 : \frac{(Q,[(F_X \to F_Y)]=[\overline{F_X}])}{(Q,[(F_X \to F_Y)],[\overline{F_Y}]=[\top])} \quad 5 : \frac{(Q,[(F_X \to F_Y)]=[\overline{F_X}])}{(Q,[(F_X \to F_Y)],[F_X]=[\overline{F_Y}])} \quad 6 : \frac{(Q,[(F_X \to F_Y)]=[\top])}{(Q,[(F_X \to F_Y)],[\overline{F_X}]=[\top])}$$

$$7 : \frac{(Q,[(F_X \to F_Y)]=[\top])}{(Q,[(F_X \to F_Y)],[F_Y]=[\top])} \quad 8 : \frac{(Q,[(F_X \to F_Y)]=[\top])}{(Q,[(F_X \to F_Y)],[F_X]=[F_Y])} \quad 9 : \frac{(Q,[(F_X \to F_Y)]=[F_Y])}{(Q,[(F_X \to F_Y)],[F_X]=[\top])}$$

The semantic link between the premise and the conclusion of a fusion rule is a model preservation equivalence link which will be shown in the next section.

We are now able to defined what is a deduction tree and what is a proof in the S_{QBF} system.

Definition 3 (Deduction tree). *A* deduction tree *is defined by induction as follows:*

- *every non explicitly contradictory formula relation is a deduction tree for the S_{QBF} system;*
- *if R is a formula relation, r a rule of the S_{QBF} system such that the premise is satisfied by R and that the result R', unique conclusion of the application of the rule on the premise, is not explicitly contradictory and if ∇' is a deduction tree of root R' then $\frac{\nabla'}{R}r$ is a deduction tree of root R;*
- *if R is a formula relation, r a rule of the S_{QBF} system such that the premise is satisfied by R and that the results R' and R'', conclusions of the application of the rule on the premise are not explicitly contradictory, and if ∇' and ∇'' are deduction trees of root respectively R' and R'' then $\frac{\nabla' \quad \nabla''}{R}r$ is a deduction tree of root R.*

Now we are able to define what is a proof for a QBF in the S_{QBF} system.

Definition 4 (Axiom and proof). *An* axiom *is a formula relation not explicitly contradictory that contains only two classes and such it is impossible to build a deduction tree with the axiom as root and containing a formula relation explicitly contradictory. A* proof *for a QBF QM in the S_{QBF} system is a deduction tree with $Id_{QM}([M] = [\top])$ as root such that every leaf is an axiom.*

Axiom definition prevent from contradiction in a class. This definition is only reduced to the first condition if quantifier elimination rules are only used if no other rule can be applied (since then contradiction has necessarily already be deduced from the rules).

The formula relation $(\varepsilon, [(a \to b), a, \overline{b}, \top], [\overline{(a \to b)}, \overline{a}, b, \overline{\top}])$ (we exceptionally explicit all the classes) is not explicitly contradictory but is not an axiom since:

$$\frac{(\varepsilon, [(a \to b), a, \overline{b}, \top, \overline{(a \to b)}, \overline{a}, b, \overline{\top}])}{(\varepsilon, [(a \to b), a, \overline{b}, \top], [\overline{(a \to b)}, \overline{a}, b, \overline{\top}])}\,1$$

This forbids a proof which uses only the elimination rule for existential quantifier like this:

$$\cfrac{\cfrac{(\varepsilon, [(a \to b), a, \overline{b}, \top], [\overline{(a \to b)}, \overline{a}, b, \overline{\top}])}{(\exists b, [(a \to b), a, \top], [\overline{(a \to b)}, \overline{a}, \overline{\top}], [b], [\overline{b}])}}{(\exists a \exists b, [(a \to b), \top], [\overline{(a \to b)}, \overline{\top}], [a], [\overline{a}], [b], [\overline{b}])}$$

Figure 3.2 develops the proof tree of the example of the introduction.

Fig. 2. Proof tree for QBF $\forall a \forall b \exists c M$ with $M = (((a \to c) \to \neg(c \to a)) \to \neg((b \to c) \to \neg(c \to b)))$, $F_0 = (a \to c)$, $F_1 = (c \to a)$, $F_4 = (c \to b)$, $F_3 = (b \to c)$, $F_2 = (F_0 \to \overline{F_1})$, $F_5 = (F_3 \to \overline{F_4})$ and $M = (F_2 \to \overline{F_5})$. Fusioned classes appear in the upper part of the sequent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\nabla_\bot}{(\forall b\exists c; [M,\top,\overline{a}] = [F_0],[F_1,\overline{c}])}{}^{9}}{(\forall b\exists c; [M,\top,\overline{a}],[F_1] = [\overline{c}])}{}_{6}}{(\forall b\exists c; [M,\top] = [\overline{a}])}{}_{4} \quad \dfrac{\dfrac{\dfrac{\nabla_\top}{(\forall b\exists c; [M,\top,a,F_1],[F_0] = [c])}{}^{4}}{(\forall b\exists c; [M,\top,a] = [F_1])}{}_{9}}{(\forall b\exists c; [M,\top] = [a])}{}_{7}}{(\forall a\forall b\exists c; [M] = [\top])}\,\forall$$

with $\nabla_\bot$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\varepsilon; [F_3,M,\top,\overline{a},F_0,\overline{b},\overline{F_5},F_4,F_1,\overline{c},\overline{F_2}])}{(\exists c; [F_3,M,\top,\overline{a},F_0,\overline{b}] = [\overline{F_5},F_4,F_1,\overline{c},\overline{F_2}])}{}^{\exists\bot}}{(\exists c; [F_3,M,\top,\overline{a},F_0,\overline{b}],[\overline{F_5}] = [F_4,F_1,\overline{c},\overline{F_2}])}{}^{8}}{(\exists c; [F_3] = [M,\top,\overline{a},F_0,\overline{b}],[F_4,F_1,\overline{c},\overline{F_2}])}{}_{9}}{(\exists c; [M,\top,\overline{a},F_0,\overline{b}],[F_4] = [F_1,\overline{c},\overline{F_2}])}{}_{6}}{(\exists c; [M,\top,\overline{a},F_0] = [\overline{b}],[F_1,\overline{c},\overline{F_2}])}{}_{4} \quad \dfrac{\dfrac{\dfrac{\dfrac{(\exists c; [M,\top,\overline{a},F_0,b,F_4],[F_1,\overline{c},\overline{F_2},\overline{F_3},F_5])}{(\exists c; [M,\top,\overline{a},F_0,b,F_4],[F_1,\overline{c},\overline{F_2},\overline{F_3}] = [F_5])}{}^{8}}{(\exists c; [M,\top,\overline{a},F_0,b,F_4],[F_1,\overline{c},\overline{F_2}] = [\overline{F_3}])}{}_{4}}{(\exists c; [M,\top,\overline{a},F_0,b] = [F_4],[F_1,\overline{c},\overline{F_2}])}{}_{9}}{(\exists c; [M,\top,\overline{a},F_0] = [b],[F_1,\overline{c},\overline{F_2}])}{}_{7}}{(\forall b\exists c; [M,\top,\overline{a},F_0],[F_1,\overline{c}] = [\overline{F_2}])}\,\forall$$

and $\nabla_\top$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\exists c; [M,\top,a,F_1,\overline{b},F_3],[\overline{F_0},\overline{c},F_2,F_4,\overline{F_5}])}{(\exists c; [M,\top,a,F_1,\overline{b},F_3],[F_0,c,\overline{F_2},\overline{F_4}] = [F_5])}{}^{8}}{(\exists c; [M,\top,a,F_1,\overline{b}] = [F_3],[F_0,c,\overline{F_2},\overline{F_4}])}{}_{9}}{(\exists c; [M,\top,a,F_1,\overline{b}],[F_0,c,\overline{F_2}] = [\overline{F_4}])}{}_{6}}{(\exists c; [M,\top,a,F_1,\overline{b}],[F_0,c,\overline{F_2}])}{}_{4} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\varepsilon; [M,\top,a,F_1,b,F_4,F_0,c,\overline{F_2},F_3,\overline{F_5}])}{(\exists c; [M,\top,a,F_1,b,F_4] = [F_0,c,\overline{F_2},F_3,\overline{F_5}])}{}^{\exists\top}}{(\exists c; [M,\top,a,F_1,b,F_4],[F_0,c,\overline{F_2},F_3] = [\overline{F_5}])}{}^{8}}{(\exists c; [M,\top,a,F_1,b,F_4],[F_0,c,\overline{F_2}] = [F_3])}{}_{4}}{(\exists c; [M,\top,a,F_1,b] = [F_4],[F_0,c,\overline{F_2}])}{}_{9}}{(\exists c; [M,\top,a,F_1,b],[F_0,c,\overline{F_2}])}{}_{7}}{(\forall b\exists c; [M,\top,a,F_1],[F_0,c] = [\overline{F_2}])}\,\forall$$

3.3 Semantics

We introduce an interpretation function in order to explicit the semantics of a formula relation as a QBF. This interpretation allows us to prove the soundness and completeness of our sequent calculus w.r.t. the semantics of QBF.

Definition 5 (Interpretation function). *The interpretation function* $(\cdot,\cdot)^*$ *of a formula relation is a function from the formula relation set to the QBF defined as follows:*

$$(Q,P)^* = Q \bigwedge_{C\in P} (\bigwedge_{F,F'\in C} (F \leftrightarrow F')).$$

Clearly, $(Q, Id_{QM}([M] = [\top]))^* \cong QM$.

Soundness of the S_{QBF} system w.r.t. the semantics of QBF is a direct consequence of the following lemma.

Lemma 1 (Soundness of the fusion rules). *Let R' be a formula relation which is the result of the application of a fusion rule on a formula relation R then $R'^* \cong R^*$.*

For the completeness, it is clear that directly from the validity computation with the definition of the semantics of the quantifiers it is possible to build a proof in the S_{QBF} system: this proof, from bottom to top, eliminates all the quantifiers and then makes the fusion of the classes thanks to the rules 4, 6, 7 and 9.

Theorem 1 (Soundness and completeness of the S_{QBF} system w.r.t. the semantics of QBF). *A QBF QM is valid if and only if the formula relation $Id_{QM}([M] = [\top])$ has a proof for the S_{QBF} system.*

4 From QBF to QCSP(QBF)

In order to define our QCSP(QBF), whose S_{QBF} is the proof-theoretical justification, as a CSP(QBF) with a quantified search algorithm we need to define a new domain for the variables composed of a Boolean domain and an equivalence domain with operations that respect the properties of quantifier order and equivalence.

Definition 6 (Equivalence domain).

The order polarity set is the set $\mathbb{N}^{+-} = \{i^-, i^+ | i \in \mathbb{N}^*\}$. A complementary operation $\overline{(.)}$ on $\mathbb{N}^{+-}$ is defined by $\overline{i^+} = i^-$ and $\overline{i^-} = i^+$ for all $i \in \mathbb{N}^*$. A mapping $[.]$ from $\mathbb{N}^{+-}$ to $2^{\mathbb{N}^{+-}}$ is defined as follows: for all $i \in \mathbb{N}^*$, $[i^-] = \{1^-, 1^+, \ldots, (i-1)^-, (i-1)^+, i^-\}$ and $[i^+] = \{1^-, 1^+, \ldots, (i-1)^-, (i-1)^+, i^+\}$. A mapping buildEqDom from $\{\exists, \forall\} \times \mathbb{N}^*$ to $2^{\mathbb{N}^{+-}}$ is defined as follows: **if** $q = \exists$ **then** *buildEqDom*$(q,i) = [i^+]$ **else** *buildEqDom*$(q,i) = \{i^+\}$ **endif**. *The equivalence domain set is the set $DE = \{[i^-], [i^+], \{i^+\} | i \in \mathbb{N}^*\}$. An order relation*

$\prec$ on $I\!N^{+-}$ is defined as follows: for all $i \in I\!N^*$, $i^- \prec (i+1)^-$, $i^- \prec (i+1)^+$, $i^+ \prec (i+1)^-$ and $i^+ \prec (i+1)^+$. A complementary operation $\overline{(.)}$ on DE is defined as follows: for all $d \in DE$, $\overline{d} = (d \setminus \max(d)) \cup \{\overline{\max(d)}\}$.

The intuition of the equivalence domain of a variable is how the variable is connected to its equivalence class and with which polarity.

For the example of the introduction, the initial equivalence domains of the variables are:

$$\begin{aligned}&\mathrm{EqDom}(a) = \{1^+\}, \mathrm{EqDom}(b) = \{2^+\},\\ &\mathrm{EqDom}(c) = [3^+], \mathrm{EqDom}(o6) = [4^+],\\ &\mathrm{EqDom}(o0) = [5^+], \mathrm{EqDom}(o1) = [6^+],\\ &\mathrm{EqDom}(o3) = [7^+], \mathrm{EqDom}(o4) = [8^+],\\ &\mathrm{EqDom}(o2) = [9^+], \mathrm{EqDom}(o5) = [10^+].\end{aligned}$$

Definition 7 (CSP(QBF) relations and filtering operators). *The domain of a variable v is a pair $(BoolDom(v), EqDom(v))$ in $2^{\{\top,\bot\}} \times DE$. Table 1 defines four unary relations $(= \top)$, $(= \bot)$, $(\neq \top)$ and $(\neq \bot)$, two binary relations $\sim$ and $\not\sim$ and one ternary relation imp with their respective filtering operators.*

Table 1. Relations and filtering operators for CSP(QBF)

$v = K$, v a variable and $K \in \{\top, \bot\}$	$\mathrm{BoolDom}(v) = \{K\}$
$v \neq K$, v a variable and $K \in \{\top, \bot\}$	$\mathrm{BoolDom}(v) = \{\overline{K}\}$

$v_i \sim v_j$,
v_i, v_j two variables
	if $\overline{\mathrm{EqDom}(v_i)} = \mathrm{EqDom}(v_j)$ **then**
	  $\mathrm{EqDom}(v_i) = \mathrm{EqDom}(v_j) = \emptyset$
	else if $\max(\mathrm{EqDom}(v_i)) \prec \max(\mathrm{EqDom}(v_j))$ **then**
	  $\mathrm{EqDom}(v_j) = \mathrm{EqDom}(v_i)$
	else
	  $\mathrm{EqDom}(v_i) = \mathrm{EqDom}(v_j)$
	end if

$v_i \not\sim v_j$,
v_i, v_j two variables
	if $\mathrm{EqDom}(v_i) = \mathrm{EqDom}(v_j)$ **then**
	  $\mathrm{EqDom}(v_i) = \mathrm{EqDom}(v_j) = \emptyset$
	else if $\max(\mathrm{EqDom}(v_i)) \prec \max(\mathrm{EqDom}(v_j))$ **then**
	  $\mathrm{EqDom}(v_j) = \overline{\mathrm{EqDom}(v_i)}$
	else
	  $\mathrm{EqDom}(v_i) = \overline{\mathrm{EqDom}(v_j)}$
	end if

$\mathrm{imp}(v_i, v_j, v_k)$,
v_i, v_j, v_k variables or $\top$ or $\bot$
	case $v_k \not\sim v_j : v_i \neq \top, v_j \neq \top$
	case $v_i \sim v_k : v_i = \top, v_j = \top$
	case $v_k = \bot : v_i = \top, v_j \neq \top$
	case $v_j = \bot : v_k \not\sim v_i$
	case $v_i \not\sim v_j : v_k \not\sim v_i$
	case $v_i = \bot : v_k = \top$
	case $v_j = \top : v_k = \top$
	case $v_i \sim v_j : v_k = \top$
	case $v_i = \top : v_k \sim v_j$

Filtering operators maintain an equivalence domain in such a way that if the conditions of explicitly contradictory relation formula of Definition 1 are fulfilled then it becomes empty. They also apply the nine class fusion rules of Definition 2.

We are now able to define our QCSP(QBF) which is in fact a CSP(QBF) with a Quantified search algorithm defined below.

Definition 8 (QCSP(QBF)). *QCSP(QBF) is the tuple*

$$\langle \mathbf{V}, rank, quant, \mathbf{D} = \{(\{\bot, \top\}, buildEqDom(quant(v), rank(v)))|v \in \mathbf{V}\}, \mathbf{C}\rangle$$

where each propositional symbol of the QBF is associated to a variable and $\mathbf{C}$ *is a set of constraints over the variables of* $\mathbf{V}$ *from the relations expressed in Definition 7.*

Definition 9 (Quantified search algorithm).

```
In: A QBF $\phi$
Out: valid if the QBF $\phi$ is valid and not_valid otherwise
  $C := decompose(\phi)$
  while true do
    switch $reachfixpoint(C)$ do
      case failure
      backtrack to last existential propositional symbol choice
      if none then
        return not_valid
      end if
      case success
      backtrack to last universal propositional symbol choice
      if none then
        return valid
      end if
      case branch
      select next propositional symbol $x$
      select next Boolean value for propositional symbol $K$
      add to $C$ constraint $(x = K)$
    end switch
  end while
```

Values of the new domains associated to Boolean domains are not used during branching, then they do not increase the size of the search tree and they are only used during propagation.

Elimination rules of Definition 2 are ensured by the Quantified search algorithm.

Table 2 shows the execution trace of the algorithm $QCSP\_QBF$ with for QBF argument $\forall a \forall b \exists c(((a \to c) \to \neg(c \to a)) \to \neg((b \to c) \to \neg(c \to b))))$ of the example of the introduction.

Following theorem establishes the link between $QCSP\_QBF$ algorithm and S_{QBF} system.

Table 2. Execution trace of the algorithm $QCSP\_QBF$ with for QBF argument $\forall a \forall b \exists c(((a \rightarrow c) \rightarrow \neg(c \rightarrow a)) \rightarrow \neg((b \rightarrow c) \rightarrow \neg(c \rightarrow b))))$

```
DomBool(o6) <- 1
DomBool(a) <- 0
PROPAGATE
imp(c, a, o1)    // DomEq(o1) <- !DomEq(c)
imp(a, c, o0)    // DomBool(o0) <- 1
imp(o0, !o1, o2) // DomEq(o2) <- !DomEq(o1)
BRANCH
DomBool(b) <- 0
PROPAGATE
imp(c, b, o4)    // DomEq(o4) <- !DomEq(c)
imp(b, c, o3)    // DomBoolo3) <- 1
imp(o3, !o4, o5) // DomEq(o5) <- !DomEq(o4)
imp(o2, !o5, o6) // DomBool(o2) <- 0,
                 // DomBool(o5) <- 0
imp(o0, !o1, o2) // DomBool(o1) <- 1
imp(o3, !o4, o5) // DomBool(o4) <- 1
imp(c, a, o1)    // DomBool(c) <- 0
PARTIALLY SOLVED
DomBool(a) <- 0
DomBool(b) <- 1
PROPAGATE
imp(c, a, o1)    // DomEq(o1) <- !DomEq(c)
imp(a, c, o0)    // DomBool(o0) <- 1
imp(c, b, o4)    // DomBool(o4) <- 1
imp(b, c, o3)    // DomEq(o3) <- DomEq(c)
imp(o0, !o1, o2) // DomEq(o2) <- !DomEq(o1)
imp(o3, !o4, o5) // DomEq(o5) <- !DomEq(o3)
PARTIALLY SOLVED
DomBool(a) <- 1
PROPAGATE
imp(c, a, o1)    // DomBool(o1) <- 1
imp(a, c, o0)    // DomEq(o0) <- DomEq(c)
imp(o0, !o1, o2) // DomEq(o2) <- !DomEq(o0)
BRANCH
DomBool(b) <- 0
PROPAGE
imp(c, b, o4)    // DomEq(o4) <- !DomEq(c)
imp(b, c, o3)    // DomBool(o3) <- 1
imp(o3, !o4, o5) // DomEq(o5) <- !DomEq(o4)
PARTIALLY SOLVED
DomBool(b) <- 1
PROPAGATE
imp(c, b, o4)    // DomBool(o4) <- 1
imp(b, c, o3)    // DomEq(o3) <- DomEq(c)
imp(o3, !o4, o5) // DomEq(o5) <- !DomEq(o3)
imp(o2, !o5, o6) // DomBool(o2) <- 0,
                 // DomBool(o5) <- 0
imp(o0, !o1, o2) // DomBool(o0) <- 1
imp(a, c, o0)    // DomBool(c) <- 1
imp(o3, !o4, o5) // DomBool(o3) <- 1
SOLVED
```

Theorem 2. (Soundness and completness of $QCSP\_QBF$ algorithm w.r.t. S_{QBF} system) *Let QM be a QBF. $Id_{QM}([M] = [\top])$ has a proof if and only if $QCSP\_QBF(QM)$ returns* `valid`*. $Id_{QM}([M] = [\top])$ has not a proof if and only if $QCSP\_QBF(QM)$ returns* `not_valid`*.*

5 Experimental results

We have implemented the constraint equivalence in Gecode [24] which is a generic constraint development environment (i.e a toolkit for developing constraint-based systems) implemented in C++. We did not need to modify Gecode. We have implemented the equivalence domain with intervals and have disconnected all optimisations. Table 3 reports some results. First column reports the name of the benchmark from $QBFLIB\_1.0$ [14]; second column reports the binder of the benchmark; third, fourth and fifth columns report respectively the number of propagations, the number of nodes of the search tree and the number of failures for the solver *without* equivalence constraints; sixth column reports the speedup ($= \frac{\text{cpu time without}*100}{\text{cpu time with}} - 100$, ∞ means that the solver has not decide before 30 minutes) ; seventh, eighth and tenth columns report respectively the number of propagations, the number of nodes of the search tree and the number of failures for the solver *with* equivalence constraints. The main result is as follows: Solver with equivalence constraints is *always* better than solver without.

Table 3. Experimental results

		solver without			solver with equivalence constraints			
		#P	#N	#F	speedup	#P	#N	#F
`counter4_2.qbf`	$\exists^{12}\forall^{8}$	5630	556	3	10%	3911	205	3
`counter4_3.qbf`	$\exists^{16}\forall^{12}$	80326	8596	7	100%	54176	3369	7
`counter4_4.qbf`	$\exists^{20}\forall^{16}$	1237107	136431	15	190%	741592	39066	15
`counter4_5.qbf`	$\exists^{24}\forall^{20}$	19857976	2179745	31	160%	13065634	827942	31
`counter4_6.qbf`	$\exists^{28}\forall^{24}$	318342127	34935929	65	190%	203660692	11905482	65
`counter5_2.qbf`	$\exists^{15}\forall^{10}$	17102	2114	3	30%	9844	557	3
`counter5_4.qbf`	$\exists^{24}\forall^{20}$	16623813	2134043	15	320%	8610737	483418	15
`counter6_2.qbf`	$\exists^{18}\forall^{12}$	56410	8296	3	110%	24724	1497	3
`counter6_4.qbf`	$\exists^{30}\forall^{24}$	207892331	31010694	15	∞	91037164	5557142	15
`counter7_2.qbf`	$\exists^{21}\forall^{14}$	198683	32942	3	270%	63115	4017	3
`counter8_2.qbf`	$\exists^{16}\forall^{24}$	719668	131380	3	750%	161471	10829	3
`ring4_2.qbf`	$\exists^{21}\forall^{14}$	6962131	1053640	516	230%	2875645	295639	433
`ring5_2.qbf`	$\exists^{24}\forall^{16}$	26652296	4291339	614	410%	8600128	823310	523
`ring6_2.qbf`	$\exists^{27}\forall^{18}$	101449882	17590062	712	620%	25932045	2434105	621
`semaphore3_2.qbf`	$\exists^{27}\forall^{18}$	54782065	1304353	23	100%	31154191	652233	23
`semaphore_2.qbf`	$\exists^{20}\forall^{14}$	2691738	80409	23	90%	1571193	40257	23

6 Conclusion

We have presented the sequent calculus S_{QBF} for quantified Boolean formulae. This new sequent calculus is the proof system underlying the Boolean propagation based on literals and extend the underlying proof system of the Stålmarck's method. The sequent calculus S_{QBF} has been proven to be sound and complete w.r.t. the semantics of the quantified Boolean formulae.

The closest work is clearly the sequent calculus GQBF [10] for QBF in negative normal form, based of the decision procedure `qpro` [10]. This system includes the quantifier elimination rules $\exists\top$, $\exists\bot$ and $\forall$ since this is a top-down system "à la" DLL. The GQBF system is an extension of the classical sequent calculus for first-order logic [11] oriented to the elimination of connectors and not to the formula relation.

The power of filtering is the ability to retract the maximum of values from the domain that can not be part of a solution. In Boolean case, filtering may at most retract only one value per domain without detecting inconsistency. Filtering is clearly more powerful for domains of larger size and our equivalence constraint might be also applied to them. One of our future work is to adapt our constraint to such domains and we hope that performances of the solver will be also improved.

References

1. K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
2. A. Ayari and D. Basin. QUBOS: Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, pages 187–201, 2002.
3. M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, pages 285–300, 2005.
4. M. Benedetti and H. Mangassarian. Experience and Perspectives in QBF-Based Formal Verification. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:133–191, 2008.
5. P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Paraconsistent reasoning via quantified Boolean formulas, i: Axiomatising signed systems. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 320–331, 2002.
6. A. Biere. Resolve and Expand. In *Proceedings of the 7th International Confrerence on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 59–70, 2004.
7. L. Bordeaux and E. Monfroy. Beyond NP: Arc-consistency for quantified constraints. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 371–386, 2002.
8. M. Cadoli, A. Giovanardi, and M. Schaerf. Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In *Proceedings of the 5th Conference of the Italian Association for Artificial Intelligence (AIIA'97)*, pages 207–218, 1997.

9. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5:394–397, 1962.
10. U. Egly, M. Seidl, and S. Woltran. A solver for QBFs in negation normal form. *Constraints*, 14(1):38–79, 2009.
11. J.H. Gallier. *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., 1985.
12. I.P Gent, P. Nightingale, A. Rowley, and K. Stergiou. Solving quantified constraint satisfaction problems. *Journal of Artificial Intelligence*, 172(6-7):738–771, 2008.
13. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 347–363, 2001.
14. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. www.qbflib.org.
15. E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006.
16. A. Goultiaeva, V. Iverson, and F. Bacchus. A Compact Representation for Syntactic Dependencies in QBFs. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, pages 412–426, 2009.
17. H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
18. F. Lonsing and A. Biere. Nenofex: Expanding NNF for QBF Solving. In *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, pages 196–210, 2008.
19. A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT'72)*, pages 125–129, 1972.
20. G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In *International Conference on Principles and Practice of Constraint Programming*, 2004.
21. D.A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130:291–328, 2003.
22. J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
23. A. Sabharwal, C. Ansótegui, C.P. Gomes, J.W. Hart, and B. Selman. QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 382–395, 2006.
24. C. Schulte and G. Tack. Views and Iterators for Generic Constraint Implementations. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 817–821, 2005.
25. M. Sheeran and G. Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *Formal Methods in System Design*, 16:23–58, 2000.
26. I. Stéphan. Boolean Propagation Based on Literals for Quantified Boolean Formulae. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 452–456, 2006.
27. L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.
28. L. Zhang. Solving QBF with combined conjunctive and disjunctive normal form. In *National Conference on Artificial Intelligence (AAAI'06)*, 2006.

Solving Fuzzy DCSPs with Naming Games[⋆]

Stefano Bistarelli[1,2], Giorgio Gosti[1,3] and Francesco Santini[1,⋆⋆]

[1] Dipartimento di Matematica e Informatica,
Università degli Studi di Perugia,
`[bista,giorgio.gosti,francesco.santini]]@dipmat.unipg.it`
[2] Istituto di Informatica e Telematica (IIT-CNR) Pisa, Italy
`stefano.bistarelli@iit.cnr.it`
[3] Institute for Mathematical Behavioral Sciences,
University Of California, Irvine, USA
`ggosti@uci.edu`

Abstract. Constraint Satisfaction Problems (CSPs) are the formalization of a large range of problems that emerge from computer science. The solving methodology described here is based on the *Naming Game* (*NG*). The NG was introduced to represent N agents that have to bootstrap an agreement on a name to give to an object (i.e. a word). In this paper we focus on solving Fuzzy NGs and Fuzzy Distributed CSPs (Fuzzy DCSPs) with an algorithm for NGs: each word on which the agents have to agree on is associated with a preference represented as a fuzzy score. The solution is the agreed word associated with the highest preference value. The two main features that distinguish this methodology from Fuzzy DCSPs methods are that the system can react to small instance changes and and it does not require pre-agreed agent/variable ordering.

1 Introduction

This paper presents a distributed method to solve Fuzzy Constraint Satisfaction Problems (CSPs) [11, 15, 8, 9, 14] that comes from a generalization of the Naming Game (NG) model [12, 1, 10, 7].

In Fuzzy Distributed CSP (DCSP) protocols, the aim is to design a distributed architecture of processors, or more generally a group of agents, who cooperate to solve a Fuzzy CSP instantiation. In this framework, we see the problem as a dynamic system and we set the stable states of the system as the solutions to our CSP. To do this we design each agent so that it will move towards a stable local state. This system may be called "self-stabilizing" whenever the global stable state is obtained through the reinforcement of the local stable states [6]. When the system finds the stable state, the DCSP instantiation

[⋆] Research partially supported by the MIUR PRIN 20089M932N: "Innovative and multi-disciplinary approaches for constraint and preference reasoning, by CCOS FLOSS project "Software open source per la gestione dell'epigrafia dei corpus di lingue antiche", and by INDAM GNCS project "Fairness, Equità e Linguaggi."

[⋆⋆] The author is partially supported by "Programma Giovani Ricercatori", Gruppo Nazionale per il Calcolo Scientifico (GNCS).

is solved. A protocol designed in this way is resistant to damage and external threats because it can react to changes in the problem instance. Moreover, in our approach all agents have equal chance to reveal private information, for this reason this algorithm is unbiased ("fair') in respect to privacy.

The NG describe a set of problems in which a number of agents bootstrap a commonly agreed name for one or more objects. In this paper we discuss a NG generalization in which agents have individual fuzzy preferences over words. This is a natural generalization of the NG, because it models the endogenous agents's preferences and attitudes towards certain object naming system. This preferences may be driven by pragmatic or rational reasons: same words may be associated to other objects, same words may be too long or too complex, or other words may be easy to confuse. To model the agents preferences we add individual fuzzy preference levels to each word in the agents' domain. As we have discussed before, a NG can be viewed as a particular crisp CSP instance. But, if we add preference levels, the NG is no longer a crisp combinatorial problem: this new game may be interpreted as an optimization problem.

To represent the Fuzzy NG instance as a particular instance of a Fuzzy DCSP, we impose that the only solutions that optimize the Fuzzy DCSP are the ones in which all the agents connected by a communication edge share the same word as a naming. More specifically, we use fuzzy unary constraints to represent the preferences over words and crisp binary constraints that prevent the two variables of the scope to be assigned to different values (in our case, different names). This forces all the optimal solutions be the states in which all variables have the same assignment (name). Therefore, the Fuzzy NG algorithm solves DCSP with fuzzy unary constraints and crisp binary constraints.

In the algorithm we have an asymmetric interaction, in which one agent is the speaker and the other agents involved are listeners. To let this interaction occur our algorithm uses a central scheduler that randomly draws a speaker at each turn. This may be interpreted as a "central orchestrator" scheme, anyhow this central scheduler has no information on the DCSP instance, and has no predetermined agent/variable ordering. Therefore, we can refer to this scheduler as a "blind orchestrator", to evince that it does not perform any reasoning on the problem instance and that it does not effect the agent's privacy. This kind of distributed Fuzzy DSCP can be applied to deal with resource allocation, collaborative scheduling and distributed negotiation [8].

As a further novelty, in Section 4 we extend the algorithm to solve Fuzzy NG in order to solve a generic instance of a Fuzzy DCSP, that is a DCSP problem where both unary and binary constraints are associated with a fuzzy preference.

This paper extends the results of [3, 4] in which the some of the authors managed (non fuzzy) DCSP with NGs. The paper is organized as follows: in Sec. 2 we present the background on NGs and Fuzzy DSCPS, while Sec. 3 presents an algorithm that solves Fuzzy NGs. Section 4 extends the algorithm in Sec. 3 in order to solve generic Fuzzy DCSPs. Then, Sec. 5 presents the tests and the results for the Fuzzy NG algorithm. At last, Sec. 6 summarizes the related work and Sec. 7 report the conclusions and ideas about future work.

2 Background

2.1 Distributed Constraint Satisfaction Problem (DCSP)

A classical constraint can be seen as the set of value combinations for the variables in its scope that satisfy the constraint. In the fuzzy framework, a constraint is no longer a set, but rather a fuzzy set [11]. This means that, for each assignment of values to its variables, we do not have to say whether it belongs to the set or not, but how much it does so. In other words, we have to use a graded notion of membership. This allows us to represent the fact that a combination of values for the variables of the constraint is partially permitted. A Fuzzy CSP is defined as a triple $P = \langle X, D, C\rangle$, where X and D are the set of variables and their domain, as in classical CSPs, and C is a set of fuzzy constraints. A fuzzy constraint is a fuzzy relation R_V on a sequence of variables V. This relation, that is a fuzzy set of tuples, is defined by its membership function:

$$\mu_{R_V} : \prod_{x_i \in V} D_j \to [0, 1]$$

The membership function of the relation R_V indicates to what extent an assignment of the variables in V belongs to the relation and therefore satisfies the constraint [11]. In fuzzy constraints, preference 1 is the best one and preference 0 the worst one. The conjunctive combination $R_V \otimes R_W$ of two fuzzy relations R_V and R_W is a new fuzzy relation $R_V \cup R_W$ defined as

$$\mu_{R_V \cup R_W}(t) = min(\mu_{R_V}(t[V]), \mu_{R_W}(t[W]))\ t \in \prod_{x_i \in V \cup W} D_i$$

We can now define the preference of a complete assignment, by performing a conjunction of all the fuzzy constraints. Given any complete assignment t, its membership degree, also called satisfaction degree, is defined as

$$\mu_t = (\bigotimes_{R_V \in C} R_V)(t) = \min_{R_V \in C} \mu_{R_V}(t[V])$$

In the following of the paper we will use the $\bigotimes$ symbol to directly perform the composition of fuzzy constraints. Thus, the optimal solution of a fuzzy CSP is the complete assignment whose membership degree is maximum over all complete assignments, that is,

$$Sol(P) = \max_{t \in \prod_{x_i \in X} D_i} \min_{R_V \in C} \mu(t[V])$$

In DCSPs [15, 11], the main difference to a classical CSP is that every variable is controlled by a corresponding agent, meaning that this agent sets the variables value. Formally, A DCSP is a tuple $\langle X, D, C, A\rangle$, i.e. a CSP with a set A of n agents, not necessarily all different. When an agent controls more than one variable, this would be modelled by a single variable whose values are combinations of values of the original variable. It is further assumed that an agent knows the domain of its variable and all constraints involving the variable, and that it can reliably communicate with all other agents. The main challenge is to develop distributed algorithms that solve the CSP by exchanging messages among the agents.

2.2 Introduction to Naming Games

The NGs [12, 1, 10, 7] describe a set of problems in which a number of agents bootstrap a commonly agreed name for one or more objects.

The game is played by a population of n agents which play pairwise interactions in order to negotiate conventions, i.e. associations between forms and meanings, and it is able to describe the emergence of a global consensus among them. For the sake of simplicity the model does not take into account the possibility of homonymy, so that all meanings are independent and one can work with only one of them, without loss of generality. An example of such a game is that of a population that has to reach the consensus on the name (i.e. the form) to assign to an object (i.e. the meaning) exploiting only local interactions. However, as it will be clear, the model is appropriate to address all those situations in which negotiation rules a decision process (i.e. opinion dynamics, etc.) [1].

Each NG is defined by an interaction protocol. There are two important aspects of the NG: the agents randomly interact and use a simple set of rules to update their state; the agents converge to a consistent state in which all the objects of the set have a uniquely assigned name, by using a distributed social strategy.

Generally, two agents are randomly extracted at each turn to perform the role of the speaker and the listener (or hearer as used in [12, 1]). The interaction between the speaker and the listener determines the agents' update of their internal state. DCSPs and NGs share a variety of common features [3, 4].

2.3 The Communication Model

In this framework, we define a general model that describes the communication procedures between agents both in NGs and in DCSPs. The communication model consists of n agents (also called processors [5, 6]) arranged in a network.

We will use a central scheduler that at each turn randomly extracts the agents that will be interacting. Let two agents connected by an edge in the network be neighbors. We allocate a broadcast register in which only the speaker i can write and can be read by all the neighboring listeners [5]. At each interaction, the speaker broadcasts the same variable assignment (word) b_s to all the neighbors by assigning the value b_s to the broadcast register. For each edge of the *communication graph*, we allocate a register on which the listener can upload the communication outcome feedback f_{ij} using a predetermined signaling system.

The interaction scheme can be represented in three steps [3, 4]:

1. *Broadcast:* The speakers broadcast information related to the proposed assignment for the variable.

2. *Feedback:* The listeners feedback the interaction outcome to express some information about the speaker assignment by using a standardized signal system (e.g. *Success*, *Failure*).
3. *Update:* The speakers and the listeners update their state regarding the overall interaction outcome.

In this scheme we see that at each turn the agents update their state. The update reflects the interaction they have experienced. We have presented the general interaction scheme, wherein each NG and DCSP algorithm has its own characterizing protocol.

2.4 Self-Stabilizing Algorithms

The definition of *Self-stabilizing algorithm* in distributed computing was first introduced by [6]. A system is *self-stabilizing* whenever, each system configuration associated with a *solution* is an absorbing state (global stable state), and any initial state of the system is in the basin of attraction of at least one *solution.*

In a self-stabilizing algorithm, we program the agents of our distributed system to interact with their neighbors. The agents update their state through these interactions by trying to find a stable state in their neighborhood. Since the algorithm is distributed many legal configurations of the agents' states and their neighbors' states start arising sparsely. Not all of these configurations are mutually compatible, and so they form mutually inconsistent potential cliques. The self-stabilizing algorithm must find a way to make the global legal state emerge from the competition between these potential cliques. Dijkstra [6] and Collin [5] suggest that an algorithm designed in this way can not always converge, and a special agent is needed to break the system symmetry. In this paper, we show a different strategy based on the concept of random behavior and probabilistic transition function, which we discuss in Sec. 3.2.

3 An Algorithm for Fuzzy Naming Games

In this section we extend the NGs to take into account Fuzzy scores associated with words, therefore, we propose an algorithm that solves Fuzzy NGs. Since we deal with fuzzy values associated only with words, we can consider the Fuzzy NG as a particular instance of a Fuzzy DCSP $P = \langle X, D, C, A\rangle$ (see Sec. 2.1): in this case we have fuzzy unary constraints describing the preferences over the possible words and binary crisp constraints that are satisfied only if the words chosen for two neighboring agents are the same (i.e. $x = y$), in order to agree on the same word. Therefore, the algorithm presented in Sec. 3.1 can be used to solve Fuzzy DCSP with only fuzzy unary constraints. In Sec. 4 we extend the algorithm in order to solve also fuzzy binary constraints among agents, and consequently, to solve all Fuzzy DCSPs.

At each turn, the algorithm is based on two entities: a single *speaker*, which communicates in broadcast its choice on the word and the related fuzzy preference and a set of *listeners*, which are the neighboring agents: the neighbors are those agents that can directly communicate with the speaker by considering the communication network over the agents. At each turn t, an agent is drawn with uniform probability to be the speaker. In the following we describe in detail each step of the interaction scheme that defines the behavior between the speaker and the listeners: we consider three phases, *i) broadcast*, *ii) feedback* and *iii) update*. Each agent marks the element that it expects to be the final shared name.

3.1 Interaction Protocol

Broadcast The speaker s executes the broadcast protocol. The speaker selects a word b from the domain using a function F, and marks it. Then it sends the tuple $(b, c_{\{s\}}\eta[s := b])$ to all its neighboring listeners. $c_{\{s\}}\eta[s := b]$ is a fuzzy unary constraint that defined the preference of the speaker s for word b.

F is defined in the following way:

F**:** We randomly select, by using a uniform probability distribution, an element in the *top* subset of domain words which have the maximum fuzzy value.

Feedback All the listeners receive the broadcast message *i.e.*, $(b, c_{\{s\}}\eta[s := b])$ from the speaker.

Each listener, l_i, reads its preference level associated to the speaker broadcast, $c_{\{l_i\}}\eta[l_i := b]$, and it marks the word in its list consistent to the speaker broadcast. Then it choses one of the following feedbacks:

– *Failure.* If $c_{\{s\}}\eta[s := b] > c_{\{l\}}\eta[l := b]$ there is a *failure*, and the listener feedbacks **Fail(**$c_{\{l\}}\eta[l := b]$**)**.
– *Success.* If $c_{\{s\}}\eta[s := b] \leq c_{\{l\}}\eta[l := b]$, there is a *success*, the listener feedbacks **Succ**.

Update The listeners' feedback determines the update of the listeners and of the speaker. When the listener feedbacks a **Succ**, then the listener also sets its preference level equal to the speaker's preference level $c_{\{s\}}\eta[s := b]$. If the speaker receives only **Succ** feedback messages from all its listeners, then it does not need to update.

Otherwise, that is if the speaker receives **Fail(**$c_{\{l_j\}}\eta[l_j := b_j]$**)** feedback messages from k listeners (with $1 \leq k < n$), it selects the message with the worst fuzzy preference, i.e. **Fail(**$c_{\{l_w\}}\eta[l_w := b_w]$**)** such that, $\forall j,\ c_{l_w}\eta[l_w := b_w] < c_{l_j}\eta[l_j := b_j]$. Then it sends to all the listeners a **FailUpdate(**$c_{\{l_w\}}\eta[l_w := b_w]$**)** message. Thus the speaker and the listeners set their preference level to the worst fuzzy preference, $c_{\{l_w\}}\eta[l_w := b_w]$.

3.2 Theorems

In this Section we show the lemmas and theorems that lead to the convergence property of the algorithm in Sec. 3.1: we formally prove that the algorithm always terminates with best solution, i.e. the word with the highest fuzzy preference.

With Lemma 1 we state that a subset of constraints $C' \subseteq C$ has a higher fuzzy preference w.r.t. C. We say that a fuzzy constraint problem is α-consistent if it can be solved with a level of satiability of at least α (see also [2]).

Lemma 1 ([2]). *Consider a set of constraints C and any subset C' of C. Then we have $\bigotimes C \leq \bigotimes C'$.*

In Lemma 2 we relate the F function to the convergence of the algorithm with probability 1, related to the level of satisfiability of the problem.

Lemma 2. *If the F function selects only the domain elements with preference level larger then α, then the algorithm converges with probability 1, only if $Sol(P) \geq \alpha$.*

From [3, 4], if the F function chooses a random element in the word domain, then i) the algorithm converges to the same word, but this word could not be the optimal one, i.e. the word with the highest fuzzy preference. If we choose F in order to select only words with a preference greater than α, then the algorithm converges to a solution with a global preference greater than α.

With Prop. 1 and Prop. 2 we prepare the background for the main theorem of this section, i.e. Th. 1. Proposition 1 shows the stabilization of the algorithm after some time, while Prop. 2 states that the algorithm converges with a probability of 1.

Proposition 1. *For time $t \to +\infty$, the weight associated to the optimal solution is equal for all the agents, and its equal to the minimum preference level of that word.*

Proposition 2. *For any probability distribution the algorithm converges with a probability of 1.*

At last, we state that the presented algorithm always converge to the best solution of the Fuzzy NG.

Theorem 1. *Since* i) *the algorithm always converges (see Prop. 2) and* ii) *by choosing a function F according to Lem. 2, the algorithm in Sec. 3.1 always converges to best Fuzzy solution, i.e. to the solution with the highest preference possible.*

4 Solving Fuzzy Distributed Constraint Satisfaction Problems with Naming Games

In this section we adapt the Fuzzy NG algorithm given in Sec. 3 in order to solve Fuzzy DCSPs in general.

As proposed in [15], we assign to each variable $x_i \in X$ of the $P = \langle X, D, C, A \rangle$, an agent $a_i \in A$. We assume that each agent knows all the constraints that act over its variables [15]. Each agent $i = 1, 2, \ldots, n$ (where $|A| = n$) searches its own variable domain $d_i \in D$ for its variable assignment that optimizes P. The degree of satisfaction of a fuzzy constraint tells us to what extent it is satisfied (see Sec. 2.1). Otherwise stated, the goal of the game is to make the agents find an assignment of their variables that maximizes the overall fuzzy score result for the problem; fuzzy preferences of constraints are combined with *min* function (see Sec. 2.1). We restrict ourselves only to unary and binary constraints. Each agent has a unary constraint c_i with support defined over its variable $x_i \in X$; this unary constraints represent the local preference of the agents for each variable assignment $d_i \in D$. Each agent can interact only with its neighbors: we may say that the communication network is determined by the network of binary constraints, since we suppose an agent $a_i \in A$ can communicate only with the $a_j \in A$ agents sharing a binary constraint, i.e. $c_{i,j} \in C$. Any binary constraint $c_{i,j}$ returns a preference value $p \in [0,1]$ which states the combined preference over the assignment of x_i and x_j together.

The extended NG is dived into time intervals during which the agents are able to interact and share information on their variable assignments and the mutual constraints. $c_{a_i}\eta[a_i := b]$ represents the preference level of agent a_i for word b. At each turn t, an agent is drawn with uniform probability to be the speaker. As in in Sec. 3 we have a speaker s and a set of listeners l_j, each of them sharing a binary constraint with s. The phases of the algorithm are the same three as in Sec. 3: *i) broadcast*, *ii) feedback* and *iii) update*.

The two main features that distinguish this methodology from Fuzzy DCSPs methods are: the system can react to small instance changes, and it does not require pre-agreed agent/variable ordering [3, 4].

4.1 Interaction Protocol

Broadcast The speaker s executes the broadcast protocol. The speaker selects a word b from the domain using a function F_i, and marks it. Also in this algorithm we can use the F_1 or F_2 functions defined in Sec. 3.1. Then it sends the couple $(b, \bigotimes c_{\{s\}}\eta[s := b])$ to all its neighboring listeners. $c_{\{s\}}$ is the set of all the constraints whose scope contains variable s, which are all known by the speaker as one of the hypothesis in a DCSP (see Sec. 2.1).

Feedback All the listeners receive the broadcast message $(b, \bigotimes c_{\{s\}}\eta[s := b])$ from the speaker. Each listener l computes $\forall d_k. \bigotimes c_{\{s\}}\eta[s := b][l = d_k]$ (let' us call this value v_k for any chosen d_k), that is it computes the combination of the fuzzy preferences (i.e. v_k) for each d_k word, supposing that s chooses word b. Each listener sends back to s a feedback message according to the following two cases:

- *Failure.* If $\max_k(v_k) > \bigotimes c_{\{s\}}\eta[s := b]$ there is a *failure*, and the listener feedbacks **Fail**$(\max_k(v_k))$.
- *Success.* If $\max_k(v_k) \leq \bigotimes c_{\{s\}}\eta[s := b]$, there is a *success*, the listener feedbacks **Succ**.

Update As in Sec. 3.1, the feedback of the listener determines the update of the listener and of the speaker. When the listener feedbacks a **Succ**, then the listener also lower the preference level for all the v_k with a higher preference value: $\forall v_k.v_k > \bigotimes c_{\{s\}}\eta[s := b]$ then it sets $v_k = \bigotimes c_{\{s\}}\eta[s := b]$. If the speaker receives only **Succ** feedback messages from all its listeners, then it does not need to update and a different speaker will be chosen in the following of the procedure (as in Sec. 3.1).

Otherwise, that is if the speaker receives a number j of $\mathbf{Fail}(v_j)$ feedback messages from j listeners, then it sets $\bigotimes c_{\{s\}}\eta[s := b] = \max_j(v_j)$. In case of a failure message, the listener does not update anything. Theorem 2 directly extends what already proved in Sec. 3.2 for the Fuzzy NG.

Theorem 2. *Given the DCSP $P = \langle X, D, C, A\rangle$, the algorithm in Sec. 4.1 always converges to best fuzzy solution.*

5 Computer Results

In this Section we show some performance results related to the algorithm presented in Sec. 3.

5.1 Empiric Variables

To evaluate the runs we define the probability of a successful interaction at time, $P_t(succ)$, given the state of the system at that time. $P_t(succ)$ is determined by the probability that an agent is a speaker at time t, and the probability that agent's interaction is a success $P_t(succ|s = a_i)$. Since, all agents have the same probability to became a speaker, we get

$$P_t(succ) = \sum P_t(succ|s = a_i)P(s = a_i) = \sum \frac{P_t(succ|s = a_i)}{N} \tag{1}$$

The $P_t(succ|s = a_i)$ depend on the state of the agent at time t. In particular it depends on the variable assignment (or word) b selected by F, and if $c_{\{s\}}\eta[s := b] \leq c_{\{l\}}\eta[l := b]$. Given an algorithm run, at each time t we can compute $P_t(succ|s = a_i)$ over the states of all agents before that the interaction is performed. Then we can compute $P_t(succ)$ with the use of Eq. 1.

5.2 Benchmark

For our benchmark, let us define a *random Fuzzy NG instance* (RFNG). To generate such an instance, we assign to each agent the same set of names, and for each agent and each agent's name, we randomly chose a different preference level. The preference levels are randomly assigned with a uniform distribution function ranging over the interval $[0, 1]$. Then, we set the constraint network to be fully connected, in this way, any agent can speak to any agent. We call this a *completely connected RFNG instance.*

We generated 5 such random instances, with 10 agents and 10 words each. For each one of these instances, we computed using a brutal force algorithm the best preference level and the word associated to this solution. Then, we ran this algorithm 10 times on each instance. To decide when the algorithm finds the solution, a graph crawler checks the agents' marked words, and their marked words preferences. If all the agents agree on the marked variable, this means they find an agreement on the name. Then, the graph crawler checks if the shared word has a preference level equal to the best preference, in such case we conclude that the algorithm has found the optimal solution.

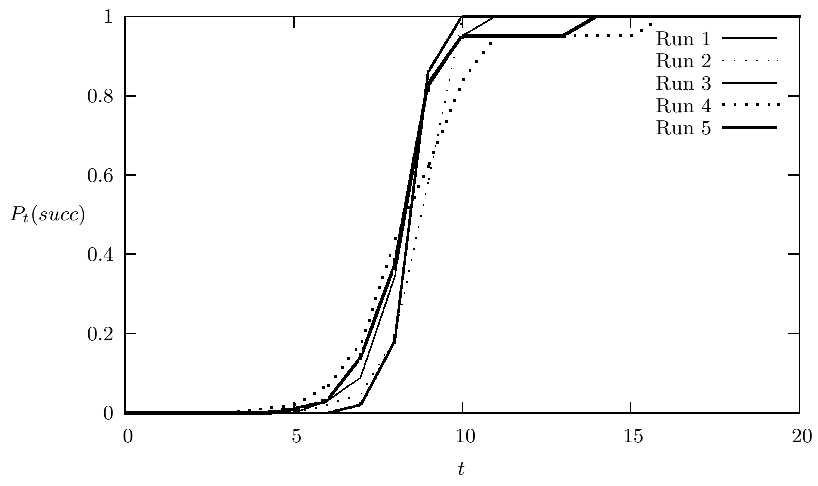

Fig. 1. Evolution of the mean $P_t(succ)$ over 5 different completely connected RFNG instances. For each instance, we computed the mean $P_t(succ)$ over 10 different runs. We set $N = 10$, and the number of words to 10.

We also define the *path RFNG instance* [4] which is a RFNG instance, in which the constraint network is a *path graph* instead of a completely connected graph, as in the completely connected RFNG instance. A path graph or linear graph is a particularly simple example of a tree, namely a tree with two or more vertices that is not branched at all, that is, contains only vertices of degree 2 and 1. In particular, it has two terminal vertices (vertices that have degree 1), while all others (if any) have degree 2.

We measured the evolution in time of $P_t(succ)$ both for the completely connected RFNG instance (Fig. 1), and the path RFNG instance (Fig. 2). When $P_t(succ) = 1$, all interactions are going to be successful, thus we are in an absorbing state, which from Th. 1, we know it is also a solution.

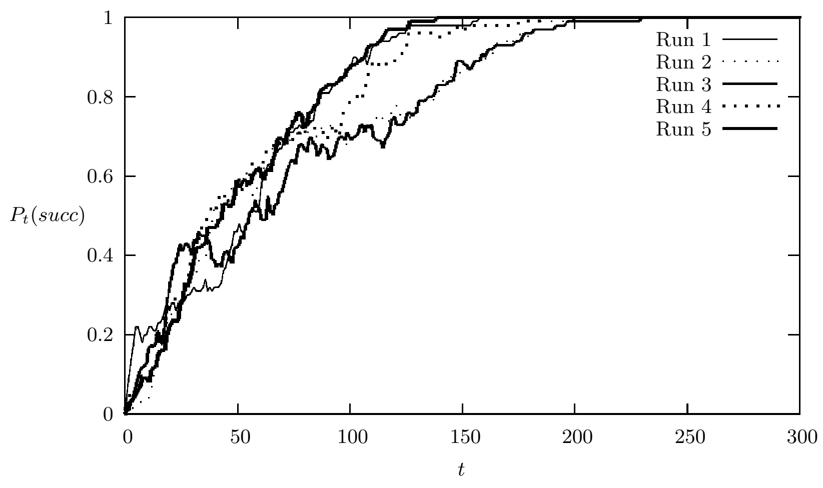

Fig. 2. Evolution of the mean $P_t(succ)$ over 5 different path RFNG instances. For each instance, we computed the mean $P_t(succ)$ over 10 different runs. We set $N = 10$, and the number of words to 10.

6 Related Work

Whilst a number of approaches have been proposed to solve DCSPs [11, 15] or centralized FCSP [11] alone, only a few work is related to the combination of DCSPs and Fuzzy CSPs.

It is important to notice the fundamental difference with the DCSP algorithms designed by Yokoo [15]. Yokoo addresses three fundamental kinds of DCSP algorithms: *Asynchronous Backtracking*, *Asynchronous weak-commitment Search* and *Distributed Breakout Algorithm* [15]. Although these algorithms share the property of being asynchronous, they require a pre-agreed agent/variable ordering. The algorithm presented in this paper does not need this initial condition.

Fuzzy DCSPs has been of interest to the Multi-Agent System community, especially in the context of distributed resource allocation, collaborative scheduling, and negotiation (e.g. [8]). Those works focus on bilateral negotiations and when many agents take part, a central coordinating agent may be required.

For example, the work in [8] promotes a rotating coordinating agent which acts as a central point to evaluate different proposals sent by other agents. Hence the network model employed in those work is not totally distributed. Another important note is that these work focuses on competitive negotiation where agents try to outsmart each other as opposed to collaborative negotiation hence does not use techniques from DCSP algorithms.

In [13, 14] the authors define the fuzzy GENET model for solving binary FCSPs. Fuzzy GENET is a neural network model for solving binary FCSPs. Through transforming FCSPs into $[0, 1]$ integer programming problems, they display the equivalence between the underlying working mechanism of fuzzy GENET and the discrete Lagrangian method. Benchmarking results confirm its feasibility in tackling CSPs and flexibility in dealing with over-constrained problems.

In [9] the authors propose two approaches to solve these problems: An iterative method and an adaptation of the *Asynchronous Distributed constraint OPTimization* algorithm (*ADOPT*) for solving Fuzzy DCSP. They also present experiments on the performance comparison of the two approaches, showing that ADOPT is more suitable for low density problems (density = num of links / number of agents).

7 Conclusions and Future Work

In this paper we have shown an algorithm to solve an extension of NG problems [12, 1, 10, 7] with fuzzy preferences over words and we have also extended this algorithm in order to solve a generic instance of a Fuzzy DCSP [11, 15, 8, 9, 14].

In the study, of such an algorithm we try to fully exploit the power of distributed calculation. Our algorithm is based on the random exploration of the system state space: it travels through the possible states until it finds the absorbing state, where it stabilizes. These goals are achieved through the union of new topics addressed in statistical physics (the NG), and the abstract framework posed by constraint solving.

This algorithm answers to an important question, can a distributed uniform probabilistic algorithm solve general Fuzzy DCSP instances? In other words, we show that a Fuzzy DCSP algorithm may work without a predetermined agent ordering, and can probabilistically solve instances that where not thought to be solvable by such algorithms. Moreover, in the real world, a predetermined agent ordering may be a quite restrictive assumption. For example, we may consider our agents to be corporations, regions in a nation, states in a federation, or independent government agencies. In all of these case a predetermined order may not be acceptable for many reasons. Hence, it is very important to explore and understand how such distributed systems may work and what problems may exist.

In future work, we intend to evaluate an asynchronous version of this algorithm in depth, and to test it using comparison metrics, such as communication cost (number of message sent), NCCCs (number of non-concurrent constraint checks). And then compare our algorithm, against other distributed and asynchronous algorithms, such as the distributed stochastic search algorithm (DSA), and the distributed breakout algorithm (DBA). We also intend to investigate the "fairness" in the loss of privacy between algorithms with no pre-agreed agent/variable ordering, and algorithms with pre-agreed agent/variable ordering.

Furthermore, we will try to generalize it to generic semiring-based CSP instances [2], and not only Fuzzy CSPs.

We also plan to develop other functions used to select the speaker agent in the broadcast phase of the algorithm, and to study the their convergence comparing the performance with the function F used in this paper (see Sec. 3.1).

References

1. A. Baronchelli, M. Felici, E. Caglioti, V. Loreto, and L. Steels. Sharp transition towards shared vocabularies in multi-agent systems. *CoRR*, abs/physics/0509075, 2005.
2. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *Lecture Notes in Computer Science*. Springer, 2004.
3. S. Bistarelli and G. Gosti. Solving CSPs with naming games. In A. Oddi, F. Fages, and F. Rossi, editors, *CSCLP*, volume 5655 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2008.
4. S. Bistarelli and G. Gosti. Solving distributed CSPs probabilistically. *Fundam. Inform.*, 105(1-2):57–78, 2010.
5. Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *IJCAI*, pages 318–324, 1991.
6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, November 1974.
7. N. L. Komarova, K. A. Jameson, and L. Narens. Evolutionary models of color categorization based on discrimination. *Journal of Mathematical Psychology*, 51(6):359 – 382, 2007.
8. X. Luo, N. R. Jennings, N. Shadbolt, H. Leung, , and J. H. Lee. A fuzzy constraint based model for bilateral, multi-issue negotiations in semi-competitive environments. *Artif. Intell.*, 148:53–102, August 2003.
9. X. T. Nguyen and R. Kowalczyk. On solving distributed fuzzy constraint satisfaction problems with agents. In *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, IAT '07, pages 387–390, Washington, DC, USA, 2007. IEEE Computer Society.
10. M. A. Nowak, J. B. Plotkin, and D. C. Krakauer. The evolutionary language game. *Journal of Theoretical Biology*, 200(2):147–162, September 1999.
11. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.
12. L. Steels. A self-organizing spatial vocabulary. *Artificial Life*, 2(3):319–332, 1995.
13. J. Wong, K. Ng, and H. Leung. A stochastic approach to solving fuzzy constraint satisfaction problems. In Eugene Freuder, editor, *Principles and Practice of Constraint Programming CP96*, volume 1118 of *Lecture Notes in Computer Science*, pages 568–569. Springer Berlin Heidelberg, 1996. 10.1007/3-540-61551-2-119.
14. J. H. Y. Wong and H. Leung. Extending genet to solve fuzzy constraint satisfaction problems. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98 IAAI '98, pages 380–385, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.
15. M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3:185–207, June 2000.

An Algebra of Search Spaces

Martin Brain

University of Bath

Abstract. This paper presents an overview of an ongoing program of research aiming to produce a representation-independent formalism for reasoning about search problems and the algorithms used to solve them. Unlike previous approaches, the focus of this work is on formalising the notion of search space rather than just the parts of the space explored by given algorithms. Algebraic structures, I-Spaces, are used to represent search spaces so that the results are syntax and language independent. The basic definition of I-Spaces, a model of computation and an algebraic semantics for CSPs are given as an illustration of this approach.

One of the consequences of the existence of $\mathcal{NP}$-Complete problems is that it is possible to think of 'universal' algorithms that can solve any $\mathcal{NP}$ problem. Thus declarative $\mathcal{NP}$ programming systems in which the user simply describes the problem using a declarative modelling language and the system automatically computes solutions, are conceivable. CSP [4], SMT, SAT [3] and ASP [2] solvers can all be seen as examples of this kind of tool, with varying input languages.

As each of these approaches has strengths it is likely that future declarative $\mathcal{NP}$ problem solving systems and modelling languages will draw aspects from each. However this presents certain practical problems. Despite using many of the same ideas and algorithms, each field has its own formalisms, notation and bodies of theory. Mapping between languages is possible but due to the heavily syntax oriented nature of many formalisms and results, doing so becomes an exercise in symbolic manipulation and obscures the fundamental links and conceptual similarities between fields. For example, questions such as "When can an *AnsProlog* program be considered to be arc consistent?" and "How does induced width relate to proof complexity and how can they be applied to SMT?" should be straightforward questions and many experienced practitioners will be able to outline how the concepts should apply. However proving formal results of this nature is difficult and unnecessarily complex. Thus prospective designers of new modelling languages are faced with a dilemma – either design a language based on an existing formalism or re-invent numerous wheels.

This paper outlines a new formalism for understanding both modelling languages and the algorithms that process them.

1 Motivation

The top level aim of this work is the development of new modelling languages for $\mathcal{NP}$ problem solving systems. Within this program it is worth considering the three aims of developing the formalism described in this paper:

1. Develop a representation independent way of talking about search and search problems. Clearly this will allow the formalism to be applied to new modelling languages. However it also has a number of other benefits. It allows existing results about different languages to be unified, thus simplifying, clarifying and unifying the study of search problems. It also increases the 'portability' of these results and the ease of transferring concepts between modelling languages.
2. Develop a formalism that can describe both problems and the algorithms that can be used to solve them. Critically this allows implementation (and to some degree algorithm) independent methods of assessment and comparison. This provides a route to more scientific and detailed analysis of computation beyond simple time based benchmarking. Micro-benchmarking of individual solver components and the development of absolute metrics of performance should also be possible.
3. Finally, for the development of future systems is it vital to have a formalism in which ideas of problem structure can be easily explored. The performance of solvers on 'structured' problems is disproportionately better than randomly generated problems of the same size. This raises many questions; what is structure, why is it important and how can it be used? Ultimately these are steps towards understanding problem difficulty.

Given the profusion of different formalisms for understanding modelling languages and aspects of $\mathcal{NP}$ search problems, it is also worth considering why the approach described by this paper is different and what effect these differences have.

The first aim of this approach is formalising the whole search space rather than just the section traversed by a given algorithm. Proof complexity in SAT [3], abstract solver frameworks in SAT and ASP [6] and the numerous uses of tree in CSP [4] focus on capturing structures that correspond to steps in a reasoning algorithm. They do not address the spaces that contain theses trees and DAGs. By representing the search space itself and these computational structures as subspaces, it is easy to prove things about all computations and provide general bounds.

The second key difference is in the use of abstract algebra as a starting point. Points in a search space, which can be informally thought of as "states of knowledge about the problem" are represented as elements of a set. It is not necessary to identify them as assignments to constraints, (partial) interpretations, (partial) models or any other representation specific concept. This allows the bulk of the theory to be developed focusing on the semantics of the search rather than the syntax of the language. Another advantage of this approach is that static, algebraic objects can be used to understand dynamic processes. In the literature, algebraic objects are much better studied and understood than dynamic processes, meaning there are more existing results that can be leveraged.

The rest of the paper presents an overview of this algebraic approach to understanding search problems. First, Section 2 defines I-Spaces, the basic algebraic objects that will be used and their structure. States of information about the problem are represented by lattice ordered sets, with most kinds of reasoning corresponding to homomorphisms, often closure operators. Next, Section 3 defines a simple model of computation with respect to any problem that can be understood in terms of I-Spaces. Traces of these algorithms correspond to substructures of the I-Space. Thus complexity bounds on the algorithms can be given terms of static, algebraic properties of the I-Space. Finally Section 4 shows how constraint satisfaction problems can be given an algebraic semantics using I-Spaces. As examples of the value of this approach, i-consistency, induced width and propagators are presented and understood in this generalised setting. Comparable semantics for Boolean formulae in CNF and *AnsProlog* programs also exist, allowing these concepts to be 'ported' to other languages.

This paper is an overview; because of space constraints proofs and and some intermediate technical lemmas will be omitted but are available on request from the authors.

2 An Algebra of Search Spaces

Before defining I-Spaces, it is necessary to clarify what is meant by a lattice order and how this differs from a lattice. Both are formalisations of the same basic concept; lattices are the operational realisation, while lattice orders are the relational realisation.

Definition 1. *A triple $\mathscr{L} = (L, \cap, \cup)$ where L is a set and $\cap$ and $\cup$ are operators on L is a* lattice *under the following conditions:*

$$\begin{aligned} \mathit{lattice}(\mathscr{L}) \Leftrightarrow\ & \mathit{associative}(\cap, L) \wedge \mathit{commutative}(\cap, L) \wedge \mathit{idempotent}(\cap, L) \wedge \\ & \mathit{associative}(\cup, L) \wedge \mathit{commutative}(\cup, L) \wedge \mathit{idempotent}(\cup, L) \wedge \\ & (\forall l_1, l_2 \in L \centerdot l_1 \cap (l_1 \cup l_2) = l_1) \wedge \\ & (\forall l_1, l_2 \in L \centerdot l_1 \cup (l_1 \cap l_2) = l_1) \end{aligned}$$

A pair $\mathscr{K} = (L, \leqslant)$ where L is a set and $\leqslant$ is a relation on L is a lattice order *under the following conditions:*

$$\begin{aligned} \mathit{latticeOrder}(\mathscr{K}) \Leftrightarrow\ & \mathit{partialOrder}(L, \leqslant) \wedge \\ & \forall l_1, l_2 \in L \centerdot \exists l_3 \in L \centerdot \mathit{supremum}(l_3, l_1, l_2) \wedge \\ & \forall l_1, l_2 \in L \centerdot \exists l_3 \in L \centerdot \mathit{infimum}(l_3, l_1, l_2) \end{aligned}$$

These two definitions are equivalent in the sense that for every lattice there is a corresponding lattice order ($l_1 \leqslant l_2 \Leftrightarrow l_1 \cap l_2 = l_1$) and for every lattice order there is a corresponding lattice ($l_1 \cap l_2 = l_3 \Leftrightarrow infimum(l_3, l_1, l_2)$ and $l_1 \cup l_2 = l_3 \Leftrightarrow supremum(l_3, l_1, l_2)$). However the notions of homomorphism and subspace they generate are subtly different. Every lattice homomorphism is a lattice order homomorphism and every sub lattice is a sub lattice order but the converse is not true. The more permissive conditions given by lattice orders will turn out to be vital for capturing the notion of reasoning.

Having discussed what is meant by a lattice order, it is possible to define *I-Spaces*, the algebraic objects used to model search spaces.

Definition 2. *A quadruple* $\mathbb{I} = (L, \leqslant, \Phi, S)$ *is an an* I-Space *under the following conditions:*

$$\begin{aligned} ISpace(\mathbb{I}) \Leftrightarrow\ & latticeOrder(L, \leqslant) \wedge \\ & (\Phi \in L) \wedge (\forall I \in L \centerdot I \leqslant \Phi) \wedge \\ & (S \subseteq L \setminus \{\Phi\}) \wedge (\forall I_s \in S \centerdot {}^{\uparrow}I_s \setminus \{\Phi\} \subseteq S) \end{aligned}$$

At this stage they are purely mathematical objects and thus the sets and relations do not 'mean' anything. However it may be useful to consider the intuition behind them. Informally, the set L may be thought of as the set of possible states of information about the problem; partial assignments, partial interpretations or partial answer sets. The ordering, $\leqslant$, is simple inclusion; one state is less than the other if its information is contained in the larger state. Φ is the point at the top of the lattice corresponding to a contradiction or invalid assignment. Finally S is the set of solutions and upwardly bounded excluding Φ (in many applications elements of S are lower neighbours of Φ making this condition redundant).

For the purposes of reasoning about $\mathcal{NP}$ modelling languages it is sufficient to consider I-Spaces over finite sets. However extensions to the infinite case are possible and allow modelling of (non-)linear programming and mixed integer programming systems.

2.1 Internal Structure

Given the definition of I-Spaces it is possible to define properties of sets of elements of the I-Space (states of information). For example whether a state forms part of solutions and which states have to be explored to be sure of finding solutions can be expressed as internal structure of an I-Space.

Paths are a simple piece of structure within I-Spaces. They can be thought of as a set of points that have to be passed through on the Hasse diagram to get from one element to the other.

Definition 3. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$ *and a pair of elements* $I_1, I_2 \in L$ *with* $I_1 \leqslant I_2$. $\delta \subseteq L$ *is a* path *from* I_1 *to* I_2 *in* $\mathbb{I}$*:*

$$\begin{aligned} semiPath(\delta, I_1, I_2, \mathbb{I}) &\Leftrightarrow totalOrder(\delta, \leqslant|_\delta) \wedge (min(\delta) = I_1) \wedge (max(\delta) = I_2) \\ path(\delta, I_1, I_2, \mathbb{I}) &\Leftrightarrow semiPath(\delta, I_1, I_2, \mathbb{I}) \wedge \\ & \quad \nexists \delta' \subseteq L \centerdot \delta \subsetneq \delta' \wedge semiPath(\delta', I_1, I_2, \mathbb{I}) \end{aligned}$$

The length of a path is given by:

$$length(\delta) = |\delta| - 1$$

A structure that is closely tied to I-Spaces is the colour of elements. An element is said to be *green* if it is in the downset of a solution; i.e. there is at least one solution consistent with that set of information. If it is not green then it is *red*.

Definition 4. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$*, an element* $I \in L$ *is* green *or* red *in* $\mathbb{I}$ *under the following condition:*

$$\begin{aligned} green(I, \mathbb{I}) &\Leftrightarrow {}^{\uparrow}I \cap S \neq \emptyset \\ red(I, \mathbb{I}) &\Leftrightarrow \neg green(I, \mathbb{I}) \end{aligned}$$

Although the definition is simple, there are a number of subtleties that must be considered. Firstly, colour is not consistency. All green elements will be necessarily be consistent (if the elements of the I-Space are partial assignments to sets of variable – see Section 4) but the converse does not hold; consistency is a strictly weaker condition. Next it is important to note that colour is a property of the mathematical structure, not a computational property. Determining the colour of an element is $\mathcal{NP}$-Hard. This means it is not suitable as a computational step, but it is a useful theoretical concept.

Given the link between determining the colour of an element and finding the solutions of a problem, a natural question to ask is what the minimal set of points whose colour has to be identified to determine the colour of a given point? Clearly finding the colour of every point in the upper set is sufficient. However, smaller sets also exist. These sets are *arches*.

Definition 5. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$ *and* $I \in L$*,* $A \subset L$ is an arch over I (in $\mathbb{I}$) *when the following condition holds:*

$$\begin{aligned} leaf(I, \mathbb{I}) &\Leftrightarrow {}^{\uparrow}I = \{I, \Phi\} \\ &\Leftrightarrow I \in lowerNeighbours(\Phi) \\ arch(A, I, \mathbb{I}) &\Leftrightarrow \forall I' \in {}^{\uparrow}I \setminus \{\Phi\} \,\bullet\, \exists I_A \in A \,\bullet\, (I_A \cup I') < \Phi \\ &\Leftrightarrow \forall I' \in {}^{\uparrow}I \,\bullet\, leaf(I', \mathbb{I}) \Rightarrow \exists I_A \in A \,\bullet\, I_A \leqslant I' \end{aligned}$$

Readers who like visual metaphors for mathematics may like to think of a classical vaulted ceiling; the point is the top of a column, the arch is the stone arch between it and the neighbouring columns, the ceiling is the upper set of the arch and Φ is the top of the roof. Within the building it is possible to get higher than the top of the columns but there is no strictly ascending route to the top of the roof that does not meet the ceiling at some point.

One consequence of the definition of arches is that there are a number of "counter intuitive" arches. For example, given a point $I_1 \in L$, $\{I_1\}$ is an arch. Likewise any element $I_2 \in L$ such that $I_2 \leqslant I_1$ will form an arch $\{I_2\}$ for I_1. To exclude these degenerate cases, a number of kinds of arch are defined.

Definition 6. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$*,* $I \in L$ *and* $A \subset L$ under arches, informative arches, upper arches *and* pairwise disjoint arches *are defined as follows:*

$$\begin{aligned} underArch(A, I, \mathbb{I}) &\Leftrightarrow arch(A, I, \mathbb{I}) \wedge (A \cap ({}^{\downarrow}I \setminus \{I\}) \neq \emptyset) \\ informativeArch(A, I, \mathbb{I}) &\Leftrightarrow arch(A, I, \mathbb{I}) \wedge \neg underArch(A, I, \mathbb{I}) \wedge \\ &\quad\ (I \in A \Rightarrow leaf(I, \mathbb{I})) \wedge (\Phi \notin A) \\ upperArch(A, I, \mathbb{I}) &\Leftrightarrow arch(A, I, \mathbb{I}) \wedge A \subseteq {}^{\uparrow}I \\ pairwiseDisjointArch(A, I, \mathbb{I}) &\Leftrightarrow arch(A, I, \mathbb{I}) \wedge \\ &\quad\ \forall I_{A_1}, I_{A_2} \in A \,\bullet\, I_{A_1} \neq I_{A_2} \Rightarrow \\ &\quad\ {}^{\uparrow}I_{A_1} \cap {}^{\uparrow}I_{A_2} = \{\Phi\} \end{aligned}$$

Arches play an important role in understanding the use of branching and heuristics in solver algorithms. They also give the first major result for I-Spaces; determining the colour of elements of an arch is sufficient to determine the colour of the point they are an arch over.

Theorem 1. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$*,* $I \in L$ *and* $arch(A, I, \mathbb{I})$*:*

$$\forall I_A \in A \,\bullet\, red(I_A, \mathbb{I}) \Rightarrow red(I, \mathbb{I})$$

Also, if A *is an upper arch:*

$$red(I, \mathbb{I}) \wedge upperArch(A, I, \mathbb{I}) \Rightarrow \forall I_A \in A \,\bullet\, red(I_A, \mathbb{I})$$

2.2 Relations Between I-Spaces

Having briefly discussed the internal structure of I-Spaces, the next area of investigation is the link between I-Spaces. Following the usual conventions of abstract algebra; similarity between I-Spaces is expressed using *homomorphisms*.

Definition 7. *Given I-Spaces* $\mathbb{I}_1 = (L_1, \leqslant, \Phi_1, S_1), \mathbb{I}_2 = (L_2, \preccurlyeq, \Phi_2, S_2)$ *a function* $h : L_1 \to L_2$ *is a* lax homomorphism *or a* homomorphism *under the following conditions:*

$$\begin{aligned} laxHomomorphism(h, \mathbb{I}_1, \mathbb{I}_2) &\Leftrightarrow (\forall I_1, I_2 \in L_1 \,\bullet\, I_1 \leqslant I_2 \Rightarrow h(I_1) \preccurlyeq h(I_2)) \wedge \\ &\quad\ h(\Phi_1) = \Phi_2 \wedge h(S_1) \subseteq S_2 \\ homomorphism(h, \mathbb{I}_1, \mathbb{I}_2) &\Leftrightarrow laxHomomorphism(h, \mathbb{I}_1, \mathbb{I}_2) \wedge h(S_1) = S_2 \end{aligned}$$

Unsurprisingly, homomorphisms preserve various key properties and internal structure of I-Spaces, some of which are given in the next proposition:

Proposition 1. *Given I-Spaces* $\mathbb{I}_1 = (L_1, \leqslant, \Phi_1, S_1), \mathbb{I}_2 = (L_2, \preccurlyeq, \Phi_2, S_2)$*,* $I_1, I_2 \in L_1$ *and a function* $h : L_1 \to L_2$ *such that* $laxHomomorphism(h, \mathbb{I}_1, \mathbb{I}_2)$*, the following inequalities hold:*

A. $S_1 = \emptyset \Leftrightarrow S_2 = \emptyset$

B. $green(I_1, \mathbb{I}_1) \Rightarrow green(h(I_1), \mathbb{I}_2)$

C. Let $(L_1, \cap, \cup)$ *be the lattice corresponding to* $(L_1, \leqslant)$ *and* $(L_2, \sqcap, \sqcup)$ *be the lattice corresponding to* $(L_2, \preccurlyeq)$*, then:*

$$h(I_1 \cap I_2) \preccurlyeq h(I_1) \sqcap h(I_2) \preccurlyeq h(I_1) \sqcup h(I_2) \preccurlyeq h(I_1 \cup I_2)$$

D. $path(\delta, I_1, I_2, \mathbb{I}_1) \Rightarrow \exists \delta' \subseteq L_2 \centerdot h(\delta) \subseteq \delta' \wedge path(\delta', h(I_1), h(I_2), \mathbb{I}_2)$

The inequalities in the third result is a direct consequence of the use of lattice orders in the definition of I-Spaces. A stronger result would not allow all forms of reasoning to be captured. For example the information that can be inferred from $a \wedge b$ by some reasoning technique should contain the information inferred from a and the information inferred from b but may contain additional information. Requiring that $h(I_1) \sqcup h(I_2) = h(I_1 \cup I_2)$ would preclude any kind of reasoning with this property from being a homomorphism.

Given the definition of homomorphisms, *isomorphisms* can be defined in the usual fashion.

Definition 8. *Given I-Spaces* $\mathbb{I}_1 = (L_1, \leqslant, \Phi_1, S_1), \mathbb{I}_2 = (L_2, \preccurlyeq, \Phi_2, S_2)$ *a function* $i : L_1 \rightarrow L_2$ *is a* isomorphism *if it meets the following criteria:*

$$\begin{aligned} isomorphism(i, \mathbb{I}_1, \mathbb{I}_2) \Leftrightarrow\ & bijection(i, L_1, L_2) \wedge \\ & laxHomomorphism(i, \mathbb{I}_1, \mathbb{I}_2) \wedge \\ & laxHomomorphism(i^{-1}, \mathbb{I}_2, \mathbb{I}_1) \end{aligned}$$

Another use of homomorphisms is in the definition of *inference functions*, a mathematical tool used to model a variety of reasoning algorithms and strategies. The idea of inference or reasoning used is very general; it is *a deterministic method of adding information that necessarily holds if the point in question is part of a solution*. It is important to note that inference functions are intended to capture the *effect* of reasoning, not the *mechanism*.

Definition 9. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$ *a function* $e : L \rightarrow L$ *is a* inference function *if it meets the following criteria:*

$$\begin{aligned} inferenceFunction(e, \mathbb{I}) \Leftrightarrow\ & laxHomomorphism(e, \mathbb{I}, \mathbb{I}) \wedge \\ & closureOperator(e, (L, \leqslant)) \end{aligned}$$

For convenience these can be reduced to four conditions:

Extensive $\forall I \in L \centerdot I \leqslant e(I)$
Increasing $\forall I_1, I_2 \in L \centerdot I_1 \leqslant I_2 \Rightarrow e(I_1) \leqslant e(I_2)$
Conserving $e(S) \subseteq S$
Idempotent $e^2 = e$

The set of all inference functions defined on I-Space $\mathbb{I}$ *is given the following notation:*

$$E(\mathbb{I}) = \{e : L \rightarrow L | inferenceFunction(e, \mathbb{I})\}$$

The use of closure operators to model reasoning is not novel [7, 1]. However by requiring them to be lax homomorphisms it is possible to tie them to the model of search spaces given by I-Spaces.

Inference functions have a number of interesting properties, one of which is that the set of inference functions over an I-Space forms an I-Space. To formalise this result it is necessary to define notions of maximum, minimum, meet and join for inference functions.

Proposition 2. *Given an I-Space $\mathbb{I} = (L, \leqslant, \Phi, S)$ the following inference functions can be constructed:*

A. $id : L \to L$

$$id(I) = I$$

$\Rightarrow inferenceFunction(id, \mathbb{I})$

B. $perfect : L \to L$

$$perfect(I) = \Phi \cap \bigcap_{I_s \in S \cap \uparrow I} I_s$$

$\Rightarrow inferenceFunction(perfect, \mathbb{I})$

C. $\cap : E(\mathbb{I}) \times E(\mathbb{I}) \to E(\mathbb{I})$

$$(e_1 \cap e_2)(I) = e_1(I) \cap e_2(I)$$

is well defined and thus $inferenceFunction((e_1 \cap e_2), \mathbb{I})$

D. $\cup : E(\mathbb{I}) \times E(\mathbb{I}) \to E(\mathbb{I})$

$$(e_1 \cup e_2)(I) = \bigcup_{n \in \mathbb{Z}} (e_1 \circ e_2)^n(I)$$

is well defined and thus $inferenceFunction((e_1 \cup e_2), \mathbb{I})$

The join operator is particularly interesting as it corresponds to a common construct found in many hybrid reasoning systems, such as SMT solvers. Two separate reasoning strategies are alternated until a fixed point is reached. If the effect of these reasoning techniques correspond to inference functions then the order of application will affect only the efficiency, not the effect, as $\cup$ is both symmetric and associative.

Having defined minimum and maximum elements of $E(\mathbb{I})$ and meet and join operators, it is straight forward to show that $(E(\mathbb{I}), \cap, \cup)$ is a lattice and corresponds to a lattice order (the ordering relation turns out to be pointwise order), thus an I-Space can be constructed:

Theorem 2. *Given an I-Space $\mathbb{I} = (L, \leqslant, \Phi, S)$ and the set of inference functions defined on it, $E(\mathbb{I})$, let $bar : L \to L$ such that $\forall I \in L \,.\, bar(I) = \Phi$ and set $\preccurlyeq^*$ to the extension of $\preccurlyeq$ such that bar is the maximal element. Define:*

$$guide(\mathbb{I}) = \{e \in E(\mathbb{I}) | \forall I \in L \,.\, red(I, \mathbb{I}) \Rightarrow e(I) = \Phi\}$$
$$\mathbb{E}(\mathbb{I}) = (E(\mathbb{I}) \cup \{bar\}, \preccurlyeq^*, bar, guide(\mathbb{I}) \setminus \{bar\})$$

Then:

$$ISpace(\mathbb{E}(\mathbb{I}))$$

Although this result may seem esoteric and purely of algebraic interest, it allows the description of "meta-heuristics" such as lookahead as homomorphisms and inference functions on $\mathbb{E}(\mathbb{I})$, the I-Space of inference functions. Quite literally, they are meta-reasoning functions.

2.3 Substructure and I-Space Operators

Another important idea in the development of an algebra is that of substructure. In the case of I-Spaces there are two orthogonal notions of sub I-Space. A *sub order I-Space* is an I-Space whose points are a sub lattice order of the original, while a *sub solution I-Space* contains the same points but less solutions.

Definition 10. *Given I-Spaces $\mathbb{I} = (L, \leqslant, \Phi, S)$ and $\mathbb{I}' = (L', \preccurlyeq, \Phi', S')$, $\mathbb{I}'$ is said to be a* sub order I-Space *or a* sub solution I-Space *of $\mathbb{I}$ under the following conditions:*

$$\begin{aligned} \mathit{subOrderISpace}(\mathbb{I}', \mathbb{I}) &\Leftrightarrow \mathit{subLatticeOrder}((L', \preccurlyeq), (L, \leqslant)) \wedge \\ &\qquad (\Phi' = \Phi) \wedge (S' = S \cap L') \\ \mathit{subSolnISpace}(\mathbb{I}', \mathbb{I}) &\Leftrightarrow (L' = L) \wedge (\preccurlyeq = \leqslant) \wedge (\Phi' = \Phi) \wedge (S' \subseteq S \cap L') \end{aligned}$$

For convenience the following notation is used:

$$\begin{aligned} \mathbb{I}' \subseteq \mathbb{I} &\Leftrightarrow \mathit{subOrderISpace}(\mathbb{I}', \mathbb{I}) \\ \mathbb{I}' \sqsubseteq \mathbb{I} &\Leftrightarrow \mathit{subSolnISpace}(\mathbb{I}', \mathbb{I}) \\ \bigcirc(\mathbb{I}) &= \{\mathbb{I}' | \mathbb{I}' \subseteq \mathbb{I}\} \\ \square(\mathbb{I}) &= \{\mathbb{I}' | \mathbb{I}' \sqsubseteq \mathbb{I}\} \end{aligned}$$

Sub I-Spaces have many of the expected properties; both $(\bigcirc(\mathbb{I}), \subseteq)$ and $(\square(\mathbb{I}), \sqsubseteq)$ form bounded partial orders. It is worth noting that the image of $\mathbb{I}$ under a homomorphism, $h(\mathbb{I})$ is not guaranteed to be a sub I-Space of the co-domain I-Space (i.e. homomorphisms do not necessarily have an epi-mono factorisation). However if an inference function is used then $e(\mathbb{I}) \subseteq \mathbb{I}$, implying that $\bigcirc(\mathbb{I})$ contains an image of $\mathbb{E}(\mathbb{I})$.

One of the key uses of sub I-Spaces is in describing the space traversed by solver algorithms. Arches are a formalisation of the concept of a (minimal) set of points whose colour has to be determined in order to prove the colour of a given point. Thus a useful construct is a subspace for which every element either has an arch over it or is clearly red. Using an inference function to determine when a point is 'clearly' red, this intuition can be formalised as ordered vaulted sub I-Spaces.

Definition 11. *Given I-Spaces $\mathbb{I} = (L, \leqslant, \Phi, S)$ and $\mathbb{I}' = (L', \leqslant', \Phi', S')$ and an inference function $e : L \to L$* ordered vaulted sub I-Spaces *are defined as:*

$$orderedVaultedSubISpace(e, \mathbb{I}', \mathbb{I})$$
$$\Leftrightarrow$$
$$(\mathbb{I}' \sqsubseteq \mathbb{I}) \wedge \exists \preccurlyeq \subseteq L' \times L' \centerdot (partialOrder(L, \preccurlyeq) \wedge$$
$$\forall I' \in L' \centerdot \exists A' \subseteq L' \centerdot (informativeArch(A', e(I'), \mathbb{I}) \wedge \forall I_A \in A' \centerdot I' \preccurlyeq I_A))$$

One subtle but important point concerning the definition of vaulted sub I-Spaces is that the arches must be defined with respect to the original space; not the subspace. This allows the following results about ordered vaulted sub I-Spaces to be proved:

Proposition 3. *Given an I-Space $\mathbb{I} = (L, \leqslant, \Phi, S)$, an inference function $e : L \to L$ and an ordered vaulted sub I-Space $I' = (L', \leqslant |_{L'}, \Phi, S')$ the following holds:*

A. $I \in L' \Rightarrow (green(I, \mathbb{I}) \Leftrightarrow green(I, \mathbb{I}'))$
B. $I_\emptyset \in L' \Rightarrow (S = \emptyset \Leftrightarrow S' = \emptyset)$
C. $I_\emptyset \in L' \wedge \forall I_s \in S \centerdot leaf(I_s, \mathbb{I}) \Rightarrow (S' = S)$

These results are sufficient to consider models of computation for problems whose structures forms I-Spaces. However there are a lot of other 'obvious' algebraic constructions on I-Spaces that are useful in reasoning about search problems. For example, it is possible to define notions of product, quotient and restricted subtraction on I-Spaces. These allow the formalisation of problem decomposition, symmetry and the addition of constraints respectively.

3 Models of Computation

Having defined algebraic structures that are sufficient to model many of the aspects of search problems; the next step is to consider a simple model of computation. The details of the algorithm and the representation of the problem are not significant. All that is required is that states of information about the problem correspond to elements of an I-Space, with $I_\emptyset$, the minimum point in the I-Space, corresponding to the initial information, Φ to a clearly invalid state and elements of S to solutions.

The first step towards reasoning about these algorithms is to define *arch choice functions*; maps which associate an arch with every element of the I-Space. These can be thought of as modelling part of the behaviour of branching heuristics. They are not a complete model as the arches only specify the alternatives; not the order in which they should be explored.

Definition 12. *Given an I-Space $\mathbb{I} = (L, \leqslant, \Phi, S)$, a function $c : L \to \mathscr{P}(L)$ is an* arch choice function *given the following criteria:*

$$\begin{aligned} archChoiceFunction(c, \mathbb{I}) &\Leftrightarrow \forall I \in L \,\boldsymbol{.}\, arch(c(I), I, \mathbb{I}) \\ PIUArchChoiceFunction(c, \mathbb{I}) &\Leftrightarrow \forall I \in L \,\boldsymbol{.}\, pairwiseDisjointArch(c(I), I, \mathbb{I}) \\ &\qquad \wedge informativeArch(c(I), I, \mathbb{I}) \\ &\qquad \wedge upperArch(c(I), I, \mathbb{I}) \end{aligned}$$

The set of all arch choice functions defined on I-Space $\mathbb{I}$ is given the following notation:

$$C(\mathbb{I}) = \{c : L \to \mathscr{P}(L) | archChoiceFunction(c)\}$$

The key use of arch choice functions is in defining *generated vaults*. These are sets of points in an I-Space generated by a simple recursive construction; for each element in the set first an inference function is applied, then the choice function and the resulting set is added to the generated vault.

Definition 13. *Given an I-Space $\mathbb{I} = (L, \leqslant, \Phi, S)$, and functions $e \in E(\mathbb{I})$ and $c \in C(\mathbb{I})$ such that $PIUArchChoiceFunction(c, \mathbb{I})$ holds, let*
$V : L \times C(\mathbb{I}) \times E(\mathbb{I}) \to \mathscr{P}(L)$ be defined by:

$$V(I, e, c) = \left\{ \begin{array}{c} \{I\} \cup \{\Phi\} \cup \bigcup_{I_A \in c(e(I))} V(I_A, e, c) \;\neg(leaf(I, \mathbb{I}) \vee I = \Phi) \\ \{I, \Phi\} \; leaf(I, \mathbb{I}) \\ \{\Phi\} \; I = \Phi \end{array} \right\}$$

Then the vault generated by e and c *is given by:*

$$\mathbb{I}[e, c] = (L' = V(I_\emptyset, e, c), \leqslant |_{L'}, \Phi, S \cap L')$$

Unsurprisingly, not only are generated vaults sub I-Spaces of the original I-Space; they are also ordered vaulted sub I-Spaces. Thus any algorithm that explores a generated vault (for some pair of choice and inference functions) will find solutions if and only if the original problem has solutions.

Proposition 4. *Given an I-Space $\mathbb{I} = (L, \leqslant, \Phi, S)$ and functions $e \in E(\mathbb{I})$ and $c \in C(\mathbb{I})$ the following result holds:*

$$PIUArchChoiceFunction(c, \mathbb{I}) \Rightarrow orderedVaultedSubISpace(e, \mathbb{I}[e, c], \mathbb{I})$$

Thus let $Vaults_{\mathbb{I}} : \mathscr{P}(E(\mathbb{I})) \times \mathscr{P}(C(\mathbb{I})) \to \mathscr{P}(\bigcirc(\mathbb{I}))$ be given by:

$$Vaults_{\mathbb{I}}(E, C) = \{\mathbb{I}[e, c] | e \in E, c \in C\}$$

Algorithm 1 gives an example of the kind of algorithm which can be used to explore a generated vault. p is a function that gives a total order on the elements of the arches returned by the choice function c.

Critically the complexity of DEPTHFIRSTSEARCH is linked to the structure of the generated vault and thus to the structure of the I-Space. The key parameter in the complexity of computing a solution is the length of the longest path of red elements.

Algorithm 1 DepthFirstSearch

```
1: procedure DepthFirstSearch($I, e, c, p$)
2:     $I' = e(I)$
3:     if $leaf(I, \mathbb{I}) \vee I' = \Phi$ then
4:         return $I'$
5:     else
6:         $A = c(I')$
7:         for $i \in [1, |A|]$ do
8:             $I'' =$ DepthFirstSearch($\pi_i(A, p(A)), e, c, p$)
9:             if $I'' \neq \Phi$ then
10:                return $I''$
11:            end if
12:        end for
13:        return $\Phi$
14:    end if
15: end procedure
```

Theorem 3. *Given an I-Space* $\mathbb{I} = (L, \leqslant, \Phi, S)$*, functions* $e \in E(\mathbb{I})$*,* $c \in C(\mathbb{I})$ *and* $p \in P(\mathbb{I})$ *such that* $PIUArchChoiceFunction(c, \mathbb{I})$*,* $I \in L$*, let:*

$$\begin{array}{rl} \mathbb{I}' = \quad \mathbb{I}[e,c] & = (V(I_\emptyset, e, c), \preccurlyeq, \Phi, S') \\ pathtb(\mathbb{I}) & = \{\delta \subseteq L | path(\delta, I_\emptyset, \Phi, \mathbb{I})\} \\ pathtt(\mathbb{I}) & = \{\delta \subseteq L | path(\delta, I, \Phi, \mathbb{I}) \wedge I \in L \wedge \forall I \in \delta \centerdot red(I, \mathbb{I})\} \\ ts(x,y) & = \frac{x^y - 1}{x - 1} \\ h = \ maxPL(\mathbb{I}) & = \max_{\delta \in pathtb(\mathbb{I})} length(\delta) \\ w = maxA(c, L') & = \max_{I \in L'} |c(I)| \\ d = maxRPL(\mathbb{I}) & = \max_{\delta \in pathtt(\mathbb{I})} length(\delta) \end{array}$$

If $\forall I \in L \centerdot leaf(I, \mathbb{I}) \Rightarrow e(I) = \Phi \vee I \in S$ *then evaluating* DepthFirstSearch*(I,e,c,p) requires at most the given number of invocations of* DepthFirstSearch*:*

$$\begin{array}{r} \Phi \Rightarrow 1 \\ red(I, \mathbb{I}) \wedge I \neq \Phi \Rightarrow ts(w, d) \\ green(I, \mathbb{I}) \Rightarrow (h - (d+1)) * w^d + ts(w, d+1) \end{array}$$

This motivates the following general definition of difficulty.

Definition 14. *Given a set of I-Spaces* $\mathfrak{I}$ *the* difficulty *of the set are defined as:*

$$difficulty(\mathfrak{I}) = \min_{\mathbb{I} \in \mathfrak{I}} maxRPL(\mathbb{I})$$

4 I-Spaces As Algebraic Semantics

Having completed the basic definitions of I-Spaces and shown how algorithmic complexity relates to the length of red paths, the final step is to use I-Spaces to create an algebraic semantics for constraint satisfaction problems and so to generalise a number of well known results.

Constraint satisfaction problems will be represented in a general form as triples (U, D, Z) where U is a set of element, D is a partition of U with each partition corresponding to the range of a variable and $Z \subseteq \mathscr{P}(U)$ representing the constraints. $\Xi((U, D))$ is the set of CSPs over the same set of variables. This allows the definition of a map from CSP problem representations to I-Spaces.

Definition 15. *Given $\beta = (U, D)$ such that $partition(D, U)$ holds, let:*

$$\begin{aligned}
\Xi(\beta) &= \{(U, D, Z) | CSP((U, D, Z))\} \\
K &= \{P \subseteq U | \forall D_i \in D \centerdot |D_i \cap P| \leqslant 1\} \\
L &= K \uplus \{\Phi\} \\
\leqslant &= \{(I_1, I_2) \in K \times K | I_1 \subseteq I_2\} \uplus \{(I_1, \Phi) \in K \times \{\Phi\}\} \uplus \{(\Phi, \Phi)\} \\
S &= \{I \in K | \forall D_i \in D \centerdot |D_i \cap I| = 1\} \\
\Gamma(\beta) &= (L, \leqslant, \Phi, S)
\end{aligned}$$

and thus define $[\![.]\!] : \Xi(\beta) \rightarrow \Box(\Gamma(\beta))$ to be:

$$[\![R]\!] = (L, \leqslant, \Phi, \{I \in S | solution(I, R)\})$$

The definition comprises two parts; first $\Gamma(\beta)$ is defined. This is an I-Space derived just from the variables. Its elements are partial assignments (plus a single "invalid" point, Φ), with all complete assignments being solutions. The second part then associates elements of $\Xi(\beta)$, individual CSPs, with sub solution I-Spaces of $\Gamma(\beta)$. It is important to note that this map is surjective but not injective, giving a coarse grained notion of "representational equivalence", when two CSPs correspond to the same (or isomorphic) I-Spaces.

Given the link between problem representations and I-Space, the next step is to formalise ideas of reasoning. A *reasoning algorithm* is a function that takes a problem representation and a set of information about the problem and returns an expanded set of information. Formally this can be stated as $Alg : \Xi(\beta) \times L \rightarrow L$. Currying this function shows that reasoning algorithms are maps from representations to maps (and under very reasonable conditions homomorphisms and inference functions) on an I-Space. In the case of CSPs, only a single reasoning algorithm, *constraintCheck* will be needed. This is a formalisation of the most basic strategy for reasoning about CSPs; if a partial assignment violates one or more constraints (is inconsistent) then the assignment cannot be part of a solution (is red) and can thus be marked as invalid (the function returns Φ).

Definition 16. *Given $\beta = (U, D)$ such that $partition(D, U)$ holds, define $constraintCheck : \Xi(\beta) \rightarrow E(\Gamma(\beta)^{\perp})$ to be:*

$$constraintCheck((U, D, Z))(I) = \left\{ \begin{array}{ll} I & \text{if } I \neq \Phi \wedge consistent(I, R) \\ \Phi & \text{if } I \neq \Phi \wedge inconsistent(I, R) \\ \Phi & \text{if } I = \Phi \end{array} \right\}$$

Again, it is worth noting that *constraintCheck* is not injective; in turn giving a finer grained notion of "operational equivalence".

Although *constraintCheck* is very simple it can be used to describe many reasoning algorithms by using meta-inference functions such as lookahead and by combining with propagators. Propagators [4] are maps from $\Xi(\beta)$ to $\Xi(\beta)$. If they conserve the solutions of the problem then they give a lax homomorphism from $[\![R]\!]$ to $[\![\phi(R)]\!]$. More usefully they can be combined with a reasoning algorithm in a number of ways, each of which yields a new reasoning algorithm (and thus an inference function, given a particular CSP). Static usage of a propagator corresponds to using the propagator as a 'preprocessing' function, while dynamic usage encodes the assignments as part of a representation and then applies the propagator at each step of the search. Using these constructions it is possible to describe large number of reasoning techniques.

One much studied property of CSPs is the level of consistency that applies to a representation. Several of the key notions of consistency have a natural realisation in terms of I-Spaces. Due to space constraints, only i-consistency [4] will be presented here.

Proposition 5. *Given a CSP $R = (U, D, Z)$, its corresponding I-Space $[\![R]\!] = \mathbb{I} = (L, \leqslant, \Phi, S)$ with $L = K \uplus \{\Phi\}$ and let:*

$$\begin{aligned} E' &= \{constraintCheck(R)\} \\ C' &= \{c \in C(\mathbb{I}) | domainArchChoiceFunction(c, R)\} \end{aligned}$$

then:

$$\begin{aligned} iConsistent(R, i) \Leftrightarrow \forall \mathbb{I}' \in Vaults_{\mathbb{I}}(E', C') \bullet \nexists \delta \subseteq L \bullet \\ path(\delta, I_{\emptyset}, \Phi, \mathbb{I}') \wedge length(\delta) = i \end{aligned}$$

Here C' is the set of arch choice functions that correspond to branching by picking an assignment and having each possible assignment as an option. This set includes dynamic variable order heuristics. The result implies that i-consistency is equivalent to the non-existence of paths from $I_{\emptyset}$ to Φ of length i. Given the later part of these paths will be red, strong i-consistency can be seen as a statement about the existence of red paths and thus about the difficulty. This inspires the final result of the paper – that the width of the constraint graph (with respect to a given ordering) [4] forms an upper bound on the difficulty of the corresponding generated vault.

Theorem 4. *Given a CSP $R = (U, D, Z)$, its corresponding I-Space $[\![R]\!] = \mathbb{I} = (L, \leqslant, \Phi, S)$ with $L = K \uplus \{\Phi\}$ and an ordering $\preccurlyeq$ such that $totalOrder(\preccurlyeq, D)$, the following theorem holds:*

$$difficulty(\{\mathbb{I}[constraintCheck(R), orderedChoice(\preccurlyeq)]\}) \leqslant width(R, \preccurlyeq) + 1$$

5 Conclusion

I-Spaces and their use as an algebraic semantics for CSPs represent a novel approach to understanding search problems and algorithms that solve them. The two key innovations; using algebraic structures and modelling the whole search space rather than just the parts explored by particular algorithms can be seen to contribute to the aims of this work.

By choosing an algebraic structure as a foundation it is possible to state results and investigate structure independently of the syntax of the language. Thus new modelling languages can gain access to the existing bodies of work by simply providing an model in terms of I-Spaces. Likewise it is possible to understand the relation between different languages formalisations' of the same idea.

By modelling the whole search space and expressing the traces of solver algorithms as subspaces, it is possible to offer meaningful formal comparisons between different algorithms and different approaches to reasoning. It is also possible to prove properties of the search space (for example difficulty bounds) and thus apply them to all possible computations.

Although this paper only presents an overview of the development and application of I-Spaces, it is hopefully sufficient to show the approach's merit as a more abstract and cleaner way of reasoning about search problems.

References

1. Krzysztof R. Apt. The role of commutativity in constraint propagation algorithms. *ACM Trans. Program. Lang. Syst.*, 22:1002–1036, November 2000.
2. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, cup-address, 1 edition, 2003.
3. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artifical Intelligence and Applications*. IOS, IOS-address, 1 edition, February 2009.
4. Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.
5. Steven R. Givant and Ralph N. McKenzie, editors. *Alfred Tarski, Collected Papers*, volume 4. Birkhäuser, 1 edition, 1986.
6. Yuliya Lierler. Abstract answer set solvers. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 377–391, Berlin, Heidelberg, 2008. Springer-Verlag.
7. Alfred Tarski. Remarques sur les notions fondamentales de la méthodologie des mathématiques. *Annales de la Société Polonaise de Mathématique*, 7:270–272, 1928. Reprinted in [5].

Modelling Constraint Solver Architecture Design as a Constraint Problem

Ian Gent, Chris Jefferson, Lars Kotthoff, and Ian Miguel
{ipg,caj,larsko,ianm}@cs.st-andrews.ac.uk

University of St Andrews

Abstract. Designing component-based constraint solvers is a complex problem. Some components are required, some are optional and there are interdependencies between the components. Because of this, previous approaches to solver design and modification have been ad-hoc and limited. We present a system that transforms a description of the components and the characteristics of the target constraint solver into a constraint problem. Solving this problem yields the description of a valid solver. Our approach represents a significant step towards the automated design and synthesis of constraint solvers that are specialised for individual constraint problem classes or instances.

1 Introduction

Most current constraint solvers, such as Minion [3], are constructed to be as general as possible. They are monolithic in design, accepting a broad range of models. While this generality is convenient, it leads to a complex internal architecture, resulting in significant overheads and inhibiting efficiency, scalability and extensibility. Another drawback is that current solvers perform little analysis of an input model, so the features of an individual model cannot be exploited to produce a more efficient solving process. To mitigate these drawbacks, constraint solvers often allow manual tuning of the solving process. However, this requires considerable expertise, preventing the widespread adoption of constraint solving.

A possible solution to this problem is to automatically generate specialised constraint solvers. A given problem class or instance is analysed and the most suitable solver components identified. These components are then assembled into a solver. The artificial intelligence community has shown a lot of interest in this problem recently, especially in the context of algorithm portfolios [15, 9, 2, 4]. The solution proposed above takes the portfolio idea a step further – instead of selecting or configuring an existing solver, we aim to *synthesise* a specialised solver.

The techniques for analysing problems and identifying the most suitable solution strategies are still applicable for our approach, but in addition we are faced with the difficult problem of automatically assembling a constraint solver from a list of components and a specification of the problem it is to solve. Constraint programming itself is a natural fit for solving this configuration problem. In the remainder of this paper, we detail a way of expressing the architecture of a constraint solver in such a way that it can be solved as a constraint problem.

2 Background

Several other approaches for generating specialised constraint solvers exist. The Multi-tac [7] system configures and compiles a constraint solver for a specific set of problems. It does not synthesis a new constraint solver from a library of components, but customises a base solver. The customisations are limited to heuristics and do not affect all parts of the constraint solver.

KIDS [13] is a more general system and synthesises efficient algorithms from an initial specification. The approach is knowledge-based, i.e. the user supplies the knowledge required to generate an efficient algorithm for the specific problem. Refinements are limited to a number of generic transformation operations and again are not capable of customising all parts of a solver.

A review of the literature on analysing combinatorial problems and selecting the most efficient way of solving them is beyond the scope of this paper. Some of the most prominent systems are SATzilla [15] and CPHydra [9], which select from a portfolio of SAT and constraint solvers respectively based on the characteristics of a problem using machine learning.

The synthesis of a constraint solver from components is a configuration problem, many instances of which have been discussed in the literature. An overview can be found in the Configuration chapter of the Handbook of Constraint Programming [10].

One of the earliest approaches to solving configuration problems as constraint problems is by Mittal and Falkenhainer [8] and proposes dynamic constraint problems that introduce new variables as the requirements for configured components become known. They furthermore require special constraints that express whether a variable is still relevant to the partially solved problem based on the assignments made so far.

Sabin and Freuder [11] propose solving configuration problems as composite constraint satisfaction problems where values for variables can be constraint problems themselves. Stumptner et al. [14] introduce the constraint-based configuration system COCOS. Their system requires several extensions of the standard constraint paradigm as well. Mailharro [6] proposes a constraint formulation that integrates concepts from object-oriented programming. His approach relies on many of the concepts introduced in earlier work and infinite-sized domains for variables. Hinrich et al. [5] use object-oriented constraint satisfaction for modelling configuration problems. They then transform the constraint model into first order logic sentences and find a solution using a theorem solver.

Our approach works without the need to modify an existing off-the-shelf constraint solver and a solution gives a complete configuration of a solver. There is no need to solve a series of refined constraint problems. This is crucial for us because we are aiming to do this in the context of generating a constraint solver that is specialised to solve a particular problem more efficiently and want to keep possible overheads, such as repeatedly generating constraint models and calling a constraint solver, as minimal as possible.

3 Architecture specification

We use the generic software architecture description language GRASP [1, 12] to describe the components of a constraint solver. The advantages of using a generic architecture description language include available tools for checking architecture descriptions for consistency and that people without a background in constraint programming are able to work with it. We chose GRASP because it is being developed by a research group at our department and we are able to influence the design of the language towards meeting the requirements for modelling constraint solvers.

A full description of GRASP is beyond the scope of this paper and not necessary for our purposes. The relevant elements of the language are described below.

templates Templates are the high-level elements of the language that describe components. A single template can describe a memory manager for example. Templates may take parameters when they are instantiated to customise their behaviour further.

requires/provides Describe things a template needs and offers for other templates to use. A memory manager for example provides a facility for storing and retrieving data. This facility could be required by a variable to keep track of its domain.

properties Properties characterise components beyond the generic facilities they provide. A Boolean variable for example would have the property that the size of the domain is at most two.

checks Check statements model the interdependencies between components and restrictions of customisations of a component. A component that implements a specific constraint for example would place restrictions on the parameters it can be customised with (i.e. the variables that it constrains) by e.g. limiting the domain size.

The check statements of GRASP provide much power and flexibility. Only a small subset of this is needed to express the components of a constraint solver though. The relevant parts are explained below.

A subsetof B Asserts that set B contains all the elements of set A. It is used to ensure that a certain implementation has a specific set of properties and provides a specific set of facilities. It can also be used to ensure that an implementation does *not* have a property or facility.

A accepts B Asserts that B is accepted as A, e.g. if A is the parameter given to the implementation of a constraint and B is a variable implementation, it makes sure that the constraint can be put on variables of that type.

Apart from the components that describe the building blocks of a solver, there is a top-level meta-component that describes the problem to be solved. It specifies the types of variables and constraints needed and which constraint implementation needs to work with which variable implementation.

The description of the constraint solver consists of a library of solver components specified this way and the problem meta-component. The library of solver components is not specific to any constraint problem to be solved by the generated solver and describes all the implementation options for any solver. The problem meta-component encodes the requirements for solving a particular constraint problem and links components from the library into an actual constraint solver.

4 Constraint model

The requirements of a component naturally map to variables in a constraint problem that we want to find assignments for. The domain of each of those variables is determined by the components which provide the facility required, i.e. the possible implementations. Each implementation variable has a set of provides and properties attached to it. The set of provides is necessary because an implementation may provide more than the one main facility that would be required by another component. If a variable is assigned a value which determines its implementation, it *must* provide all the facilities and have all the properties that this implementation provides and has and it *must not* provide any other facilities or have any other properties. We therefore add constraints to ensure that a component variable has a certain property or provide if and only if it is assigned an implementation that has this property or provide.

There are several cases we need to consider for converting the check statements of GRASP into constraints. The first case is of the form `list subsetof properties/provides`. This requires a component implementation to provide a list of facilities or have a set of properties. The translation into constraints is straightforward; we simply require the things in `list` to be in the set of `properties/provides`. The second case of the form `properties/provides subsetof list`. This is the opposite of the previous case and *forbids* the properties/provides which are not listed explicitly. The translation into constraints is analogous to the previous case.

The final case deals with the `accepts`. The general requirement encoded is that if a parameter to an implementation requires a certain property or facility, the implementation of the parameter must provide it. The corresponding constraints are implications that require properties and provides of an implementation that might be used as a parameter to be set if they are set for the parameter.

4.1 Conditional variables and constraints

The variables and constraints mentioned so far are only valid at the top level, i.e. for the problem meta-component. We need additional constructs that encode the requirements that arise if a component is implemented in a certain way. The variables and constraints to encode the requirements take the same form as above, but they have prerequisites that need to be true in order for them to become relevant.

We chose an explicit representation of the prerequisites where the conditional variables encode them in their names. The names of the variables that model the requirements for an implementation of a component not at the top level are prefixed by the implementation choices for the top-level components. The constraints on these variables can be encoded as an implication, e.g. if component x is implemented as an A, its first parameter needs to have property Y. The name of the variable that models this first parameter would have a prefix that indicates that the superior component x is implemented as an A. The left-hand side of the implication is a conjunction of the implementation decisions made in the prerequisites.

4.2 Modelling language

We decided to use the modelling language of the Minion constraint solver. While it would be easier to use a more high-level language such as Essence, we need more fine-grained control over the solving process. In particular, we need to be able to specify the order of variables and values in the domains of variables to guide the solver towards the implementations we consider the most suitable ones. This enables us to analyse the constraint problem to be solved with the synthesised solver, identify the component implementations that are likely to provide the best performance and encode this in the constraint problem through the variable and value orderings.

The decision to use the Minion input language has some ramifications for the model. First, Minion only supports integer domain values and all component implementations, properties and provides must be mapped to integers. Furthermore, some of the constraints that the model uses are not provided by Minion and must be encoded with additional constraints and variables.

For the provides and the properties of each component variable, we added an auxiliary array of Boolean variables to represent the set. If the ith Boolean is set to true, the ith property or provides is present in the set. This means that two auxiliary arrays of Booleans are added for each component variable.

Almost all constraints can be encoded directly in Minion. We used the `watched-or` constraint to express that a component variable can have a property or provides if and only if it is assigned one of the implementations that have it. The conjunction to encode the conditional constraints was implemented with a `watched-and`. The only constraints which cannot directly be translated into Minion are the implications, as Minion only allows implications between a Boolean variable and a constraint. To mitigate this, we introduced channelling variables, one for each auxiliary array of Booleans that encode the properties and provides. The left hand side of the implication is linked to the channelling variable through an if and only if (`reify` in Minion) and the right hand side is connected to the channelling variable by an implication constraint (`reifyimply` in Minion).

5 Example

Consider the constraint problem below.

$$pvx + pvy = pvw + pvc6$$
$$pvx = pvz$$

The GRASP specification of a solver component library and problem meta-component that corresponds to the constraint problem are shown in Figure 1. The problem meta-component requires five variables and two constraints with certain properties and restrictions. The variable and constraint implementations impose further restrictions and may in turn require a memory manager. Constant variables have domain size 1, Boolean variables domain size 2 and discrete variables arbitrary-sized domains. A GAC sum needs to be able to remove values from the domain of the variables it constrains while a Boolean sum needs its first argument to be a Boolean (domain size 2) and its second argument to be a constant variable (domain size 1). The memory manager does not have any special properties or further requirements.

Parts of the Minion model generated from this description is shown in Figure 2. The first part shows the variables generated for the requirement `IPropVariable pvw` in the GRASP model (Figure 1). Apart from the main variable, there are auxiliary variables for the properties and the provides as well as variables which model the conditional requirements of `pvw` being implemented in a particular way.

The first section of the constraints section models the properties and requires `pvw` (or one of its requirements) will have if being implemented in a particular way. We especially refer the reader to the last couple of lines before the ... – these express what possible implementations for `pvw` would give it specific properties/provides and that some of the provides are not given by any of the candidate implementations and are therefore always not in the set (Boolean array element set to 0).

The second part of the constraints section models the check statements that affect the parameters of the sum constraint implementations. The variables for these components are not shown for space reasons, but are analogous to the variables for the `pvw` component. A pair of Minion constraints is required to model the implication of a component being implemented in a specific way, as outlined in Section 4.2. The first constraint reifies the conditions with the channelling variable while the second constraint establishes the implication between the channelling variable and the actual property or provide.

Note that some of the `check...accepts` statements are given additional properties for the variables to check in the GRASP model. Only the properties which are not explicitly given need to be checked by the generated constraints.

Minion finds a valid constraint solver architecture for the, admittedly trivial, encoded problem (the first solution to the generated constraint model) in just a couple of milliseconds.

```
architecture Solver {
  template CopyMemoryFactory() {
    provides IMemoryManager;
    provides IRawMemory;
    provides IViewHistory;
  }
  template DiscreteVariableFactory() {
    provides IPropVariable;
    provides IRemoveFromDomain;
    requires IMemoryManager domain;
    requires IMemoryManager bounds;
    property domainType = "bound";
    check domain.getProperties() subsetof [(MemoryChanges, 'Single')];
    check bounds.getProperties() subsetof [];
  }
  template BoolVariableFactory() {
    provides IPropVariable;
    provides IRemoveFromDomain;
    requires IMemoryManager bounds;
    property domainType = "bound";
    check bounds.getProperties() subsetof [(MemoryChanges, 'Single')];
    property domainSize = 2;
  }
  template ConstantVariableFactory() {
    provides IPropVariable;
    provides IRemoveFromDomain;
    property domainSize = 1;
    property domainType = "bound";
  }
  template GACSumFactory(P1, P2) {
    provides ISumEqCon;
    requires IMemoryManager m;
    check m.getProperties() subsetof [];
    check [IRawMemory] subsetof m.getInterfaces();
    check [IRemoveFromDomain, IPropVariable] subsetof P1.getInterfaces();
    check [IRemoveFromDomain, IPropVariable] subsetof P2.getInterfaces();
  }
  template BoolSumFactory(P1, P2) {
    provides ISumEqCon;
    requires IMemoryManager m;
    check m.getProperties() subsetof [];
    check [(domainSize, 2)] subsetof P1.getProperties();
    check [(domainSize, 1)] subsetof P2.getProperties();
    check [IPropVariable] subsetof P1.getInterfaces();
    check [IPropVariable] subsetof P2.getInterfaces();
  }
  template ThisProblem() {
    provides IProblem;
    requires IPropVariable pvw, pvx, pvy, pvz, pvc6;
    requires ISumEqCon scA, scB;
    check pvx.getProperties()  subsetof
        [(domainType, 'bound'), (domainSize, 2)];
    check pvy.getProperties()  subsetof
        [(domainType, 'bound')];
    check pvc6.getProperties() subsetof
        [(domainType, 'bound'), (domainSize, 1)];
    check scA.param(1) accepts (pvx  +
        [(domainType, 'bound'), (domainSize, 2)]);
    check scA.param(1) accepts (pvy  +
        [(domainType, 'bound')]);
    check scA.param(2) accepts (pvw  +
        [(domainType, 'bound'), (domainSize, 1)]);
    check scA.param(2) accepts (pvc6 +
        [(domainType, 'bound'), (domainSize, 1)]);
    check scB.param(1) accepts (pvx  +
        [(domainType, 'bound'), (domainSize, 2)]);
    check scB.param(1) accepts pvz;
  }
}
```

Fig. 1. Solver architecture description for simple constraint problem.

```
MINION 3
**VARIABLES**
DISCRETE pvw_1_domain {1..1}
BOOL pvw_1_domain_properties[3]
BOOL pvw_1_domain_provides[7]
DISCRETE pvw_1_bounds {1..1}
BOOL pvw_1_bounds_properties[3]
BOOL pvw_1_bounds_provides[7]
DISCRETE pvw_2_bounds {1..1}
BOOL pvw_2_bounds_properties[3]
BOOL pvw_2_bounds_provides[7]
DISCRETE pvw {1..3}
BOOL pvw_properties[3]
BOOL pvw_provides[7]
...
**CONSTRAINTS**
reify(watched-or({eq(pvw_1_domain, 1)}), pvw_1_domain_provides[0])
eq(pvw_1_domain_provides[6], 0)
eq(pvw_1_domain_provides[3], 0)
reify(watched-or({eq(pvw_1_domain, 1)}), pvw_1_domain_provides[1])
eq(pvw_1_domain_provides[4], 0)
eq(pvw_1_domain_provides[5], 0)
reify(watched-or({eq(pvw_1_domain, 1)}), pvw_1_domain_provides[2])
eq(pvw_1_domain_properties[2], 0)
eq(pvw_1_domain_properties[1], 0)
eq(pvw_1_domain_properties[0], 0)
reify(watched-or({eq(pvw_1_bounds, 1)}), pvw_1_bounds_provides[0])
eq(pvw_1_bounds_provides[6], 0)
eq(pvw_1_bounds_provides[3], 0)
reify(watched-or({eq(pvw_1_bounds, 1)}), pvw_1_bounds_provides[1])
eq(pvw_1_bounds_provides[4], 0)
eq(pvw_1_bounds_provides[5], 0)
reify(watched-or({eq(pvw_1_bounds, 1)}), pvw_1_bounds_provides[2])
eq(pvw_1_bounds_properties[2], 0)
eq(pvw_1_bounds_properties[1], 0)
eq(pvw_1_bounds_properties[0], 0)
reify(watched-or({eq(pvw_2_bounds, 1)}), pvw_2_bounds_provides[0])
eq(pvw_2_bounds_provides[6], 0)
eq(pvw_2_bounds_provides[3], 0)
reify(watched-or({eq(pvw_2_bounds, 1)}), pvw_2_bounds_provides[1])
eq(pvw_2_bounds_provides[4], 0)
eq(pvw_2_bounds_provides[5], 0)
reify(watched-or({eq(pvw_2_bounds, 1)}), pvw_2_bounds_provides[2])
eq(pvw_2_bounds_properties[2], 0)
eq(pvw_2_bounds_properties[1], 0)
eq(pvw_2_bounds_properties[0], 0)
eq(pvw_provides[0], 0)
eq(pvw_provides[6], 0)
reify(watched-or({eq(pvw, 1), eq(pvw, 2), eq(pvw, 3)}), pvw_provides[3])
eq(pvw_provides[1], 0)
reify(watched-or({eq(pvw, 1), eq(pvw, 2), eq(pvw, 3)}), pvw_provides[4])
eq(pvw_provides[5], 0)
eq(pvw_provides[2], 0)
reify(watched-or({eq(pvw, 3)}), pvw_properties[2])
reify(watched-or({eq(pvw, 2)}), pvw_properties[1])
reify(watched-or({eq(pvw, 1), eq(pvw, 2), eq(pvw, 3)}),
    pvw_properties[0])
...
reify(watched-and({eq(scA, 1)}), scA_1_param_1_provides_channel[3])
reifyimply(eq(scA_param_1_provides[3], 1),
    scA_1_param_1_provides_channel[3])
reify(watched-and({eq(scA, 1)}), scA_1_param_1_provides_channel[4])
reifyimply(eq(scA_param_1_provides[4], 1),
    scA_1_param_1_provides_channel[4])
reify(watched-and({eq(scA, 1)}), scA_1_param_2_provides_channel[3])
reifyimply(eq(scA_param_2_provides[3], 1),
    scA_1_param_2_provides_channel[3])
reify(watched-and({eq(scA, 1)}), scA_1_param_2_provides_channel[4])
reifyimply(eq(scA_param_2_provides[4], 1),
    scA_1_param_2_provides_channel[4])
...
**SEARCH**
**EOF**
```

Fig. 2. Excerpts of constraint model for Figure 1.

6 Limitations and future work

A limitation of the current system is that we are unable to express requirements which have a global effect on all components, such as whether to attach debug information. At present, we are unable to express this in GRASP and therefore cannot add it to the constraint model. We are planning on extending GRASP to support this.

We have found that in practice while solving the generated constraint problems for the first solution is quick, enumerating all solutions takes a long time because of the auxiliary variables which result in sets of separate solutions that specify the same constraint solver being found.

7 Conclusions

We have presented a way of encoding the configuration of the architecture of a constraint solver as a constraint problem such that a solution to the problem specifies a valid solver. This represents a major step towards automated synthesis of constraint solvers from a library of components for a given problem. Given a library and components and a problem specification, we can automatically and efficiently synthesis a constraint solver.

Modelling the architecture of a constraint solver as a standard constraint problem enables us to use off-the-shelf software to solve this complex configuration problem using tried and tested techniques. Instead of a single solver, we can easily generate all valid solvers by finding all solutions to the configuration problem instead of only the first one.

Acknowledgements

GRASP was designed by Dharini Balasubramaniam and Lakshitha de Silva. Peter Nightingale was involved with the modifications of GRASP. Lars Kotthoff is supported by a SICSA studentship. This work was supported by EPSRC grant EP/H004092/1. We thank the anonymous reviewer for their comments.

References

1. Balasubramaniam, D., de Silva, L., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Dominion: An architecture-driven approach to generating efficient constraint solvers. In: 9th Working IEEE/IFIP Conference on Software Architecture (WICSA) (2011)
2. Gent, I., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N., Nightingale, P., Petrie, K.: Learning when to use lazy learning in constraint solving. In: ECAI. pp. 873–878 (August 2010)
3. Gent, I., Jefferson, C., Miguel, I.: MINION: A fast scalable constraint solver. In: Proceedings of the Seventeenth European Conference on Artificial Intelligence. pp. 98–102 (2006)

4. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. 126(1-2), 43–62 (2001)
5. Hinrich, T., Love, N., Petrie, C., Ramshaw, L., Sahai, A., Singhal, S.: Using Object-Oriented constraint satisfaction for automated configuration generation. In: DSOM (2004)
6. Mailharro, D.: A classification and constraint-based framework for configuration. Artif. Intell. Eng. Des. Anal. Manuf. 12, 383–397 (1998)
7. Minton, S.: Automatically configuring constraint satisfaction programs: A case study. Constraints 1, 7–43 (1996)
8. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: AAAI. pp. 25–32 (1990)
9. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish Conference on Artificial Intelligence and Cognitive Science (2008)
10. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc. (2006)
11. Sabin, E.C.F.D.: Configuration as composite constraint satisfaction. Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop pp. 153–161 (1996)
12. de Silva, L., Balasubramaniam, D.: A model for specifying rationale using an architecture description language. Tech. rep., University of St Andrews (2011)
13. Smith, D.R.: KIDS - a Knowledge-Based software development system. In: Automating Software Design. pp. 483–514. MIT Press (1990)
14. Stumptner, M., Friedrich, G.E., Haselböck, A.: Generative constraint-based configuration of large technical systems. Artif. Intell. Eng. Des. Anal. Manuf. 12, 307–320 (1998)
15. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) 32, 565–606 (2008)

QuickLex - A Case Study in implementing constraints with Dynamic Triggers

Chris Jefferson
caj@cs.st-andrews.ac.uk

University of St. Andrews

Abstract. In this paper we consider two different mechanisms for improving the performance of a algorithm for lexicographic ordering of arrays, known as GACLex. Firstly we investigate a simpler algorithm which does not achieve GAC propagator. Secondly we investigate using *Dynamic Triggers*, a mechanism for changing the set of variables a constraint operates on dynamically during search. Finally we combine these two techniques, producing an algorithm much faster than either technique in isolation.

1 Background

The lexicographic ordering constraint imposes that two arrays of equal length [1] are ordered in lexicographic, or dictionary order. Frisch et. al published the GACLex algorithm [1], which propagates the lexicographic ordering constraint. The GACLex algorithm has both good theoretical and practical performance. In particular the cost of the algorithm is $O(n)$ on arrays of length n, and the algorithm has cost $O(n + c)$ when invoked c times down a single branch of search. It is not possible to improve these worst-case costs. However, we will consider some methods of improving GACLex and in particular show how *Dynamic Triggers*[3], a relatively new addition to constraint solvers, can improve the performance of propagation by an order of magnitude.

While lexicographic ordering is used in a number of situations, it has proved most valuable for implementing symmetry breaking, with many different frameworks using sets of lex constraints to break symmetry. These frameworks can require, in particular for breaking all symmetry, introducing tens of thousands of lexicographic ordering constraints.

2 Preliminary Experiments

The traditional lex propagator of Frisch et al. uses two pointers, α and β. We refer the reader to the original paper for a complete discussion of the algorithm. At the beginning of the algorithm α is at the start of the array, and β is at the end. α moves forward along the array while β moves backwards.

[1] In this paper we will only consider equal length arrays for simplicity

The value taken by α is defined to be the first index i where x_i and y_i do not both have the same single value in their domain. At this index, the variables x_i and y_i can be propagated so that $x_i \leq y_i$. Theorem 1 shows that this is actually sufficient for a correct (although not GAC) propagation algorithm.

Algorithm 1 SimpleLexPropagate

```
 1: $\alpha_O$: The last value of $\alpha$
 2: $\boldsymbol{x}, \boldsymbol{y}$: Arrays of variables
 3: $n$: Length of $\boldsymbol{x}$ and $\boldsymbol{y}$
 4: $\alpha \leftarrow \alpha_O$
 5: while $\alpha < n$ do
 6:    x[i].setMax(y[i].getMax())
 7:    y[i].setMin(x[i].getMin())
 8:    if !x[i].isAssigned() or !y[i].isAssigned() or !(x[i].value() = y[i].value()) then
 9:       return {Return and store the current value of $\alpha$}
10:    end if
11:    $\alpha \leftarrow \alpha + 1$
12: end while
13: {All variables are assigned and $\boldsymbol{x} = \boldsymbol{y}$}
```

Theorem 1. *The algorithm in Algorithm 1 implements a valid, but not GAC, algorithm for lexicographic ordering.*

Proof. There are parts to proving our algorithm implements a correct propagator. First of all it must delete no domain value which could take part in a solution. Secondly, when all variables are assigned it must reject assignments which are not solutions.

That this algorithm never deletes values which it should not is obvious, both because it performs a subset of the propagation of the GACLex algorithm and directly. The algorithm only ever performs deletions on the ith index, when the variables at all previous indices are assigned and equal. In this case, the variables at the ith index must satisfy the condition that $x[i] \leq y[i]$, to satisfy the lexicographic ordering constraint.

When all variables are assigned, then the propagator enforces that at the first index i where the variables are not equal, $x[i] < y[i]$ or that the arrays x and y take the same value. These are exactly the two cases in which the lexicographic ordering constraint is satisfied.

We will begin by comparing the performance of Algorithm 1 against the original. The Figure 1 shows the performance of the propagation algorithm both with and without *beta*, using original *Static* triggers.

This graph also shows the effect of enabling Minion's *Dynamic* triggering system, but not making use of any of their features. This shows that enabling Minion's dynamic trigger system comes at a high cost, so we must make substantial gains in performance to make enabling it worthwhile.

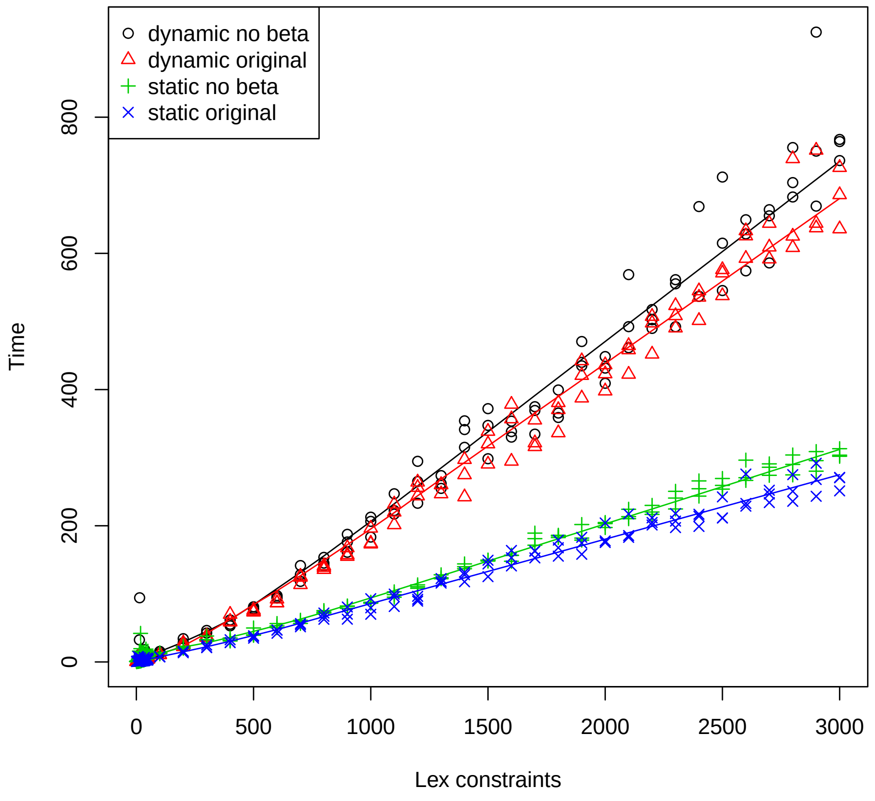

Fig. 1. Comparison of lex algorithm with and without beta

We can see from this graph that the effect of removing β is a very small decrease in performance. While the algorithm does perform slightly faster per node, the increased search space leads to a small decrease in overall performance.

3 Dynamic Triggers

Propagators operate in constraint solvers by being invoked when variables change. To improve efficiency, propagators can list only particular events on particular variables which they wish to be informed about. Dynamic triggers allow a constraint to add, remove and move triggers during search. In this section we will consider one special case of dynamic triggers, allowing constraints to remove triggers from variables they are no longer interested in. We consider 2 different techniques.

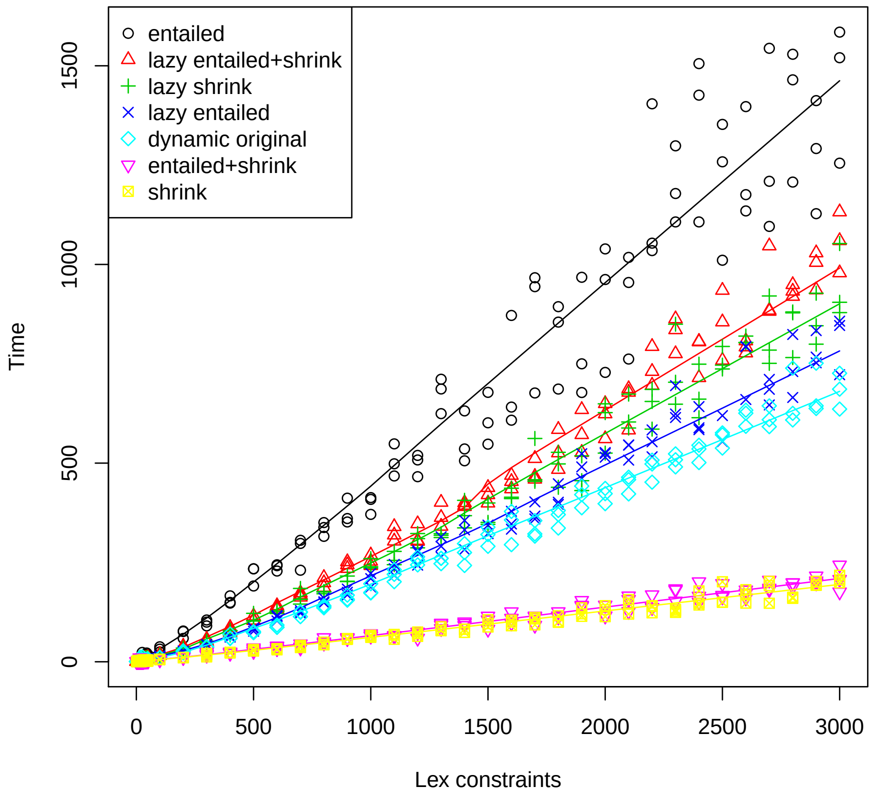

Fig. 2. Comparison of various dynamic implementations of lex

1. Remove triggers on variables after β, which are not used in the algorithm. (shrink). Note that while variables before α are also not used by the algorithm, by definition they must also be assigned, so removing the triggers from them is not useful.
2. When the constraint becomes entailed, remove triggers from all the variables (entailed).

We also consider two methods of implementing these techniques, eager and lazy. In an eager implementation we remove triggers as soon as possible. In a lazy implementation, we remove a trigger when it is invoked, and we see that it could be removed.

These results are displayed in Figure 2. This graph features many datapoints, we now point out a few interesting facts.

Eager entailment is by far the worst of the implementations considered. This is because after entailment constraints are rarely invoked again, and so removing and then re-adding all the triggers leads to a large cost.

In general it is not possible to say if lazy or eager removal of triggers is better. Lazy removal of triggers substantially improves the performance of entailment, but it impairs the performance of shrink and entail+shrink.

We see that the only algorithms which improve performance are the eager shrink and entailed+shrink, we perform almost equally.

4 QuickLex

Finally, we consider our final algorithm, quicklex. This algorithm is based on our earlier observation that when we implement lex without a *beta* pointer, the algorithm only performs work when the variables currently pointed to by the α pointer change. Therefore we shall implement a variant of lex using dynamic triggers which only watches the variables at the α pointer. Obviously these position can change over time, so the position pointed to will change too.

We shall now consider implementing our 'no beta' algorithm with dynamic triggers. We note that Figure 1 showed a loss in performance using a 'no beta' algorithm without dynamic triggers.

We can see from Figure 3 that 'quicklex' provides a substantial performance boost over both the original algorithm, and the dynamic shrinking algorithm which was the best algorithm shown previously. By placing a linear regression through out data, we find on this experiment adding each new lex constraint costs 0.012 seconds for quicklex, while it costs 0.091 seconds for the original algorithm, and 0.063 for *Shrink*, the second best algorithm we found in practice.

5 Conclusion

This paper shows a new variant of the lex algorithm, which improves the performance by an order of magnitude in practice. The most interesting feature of this algorithm is that without dynamic triggers, the new algorithm decreases performance. This shows that the performance of propagators is closely tied to the features of the solver they are run on.

In particular, these experiments were run entirely on the Minion [2] solver. The Gecode [4] solver copies the state of the problem at every node, and it is claimed by the authors that using eager entailment always improves performance, in cases where detecting entailment is free or very cheap. In future we would like to re-implement these algorithms in Gecode, to perform an in-depth comparison.

References

1. Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Propagation algorithms for lexicographic ordering constraints. *Artif. Intell.*, 170(10):803–834, 2006.

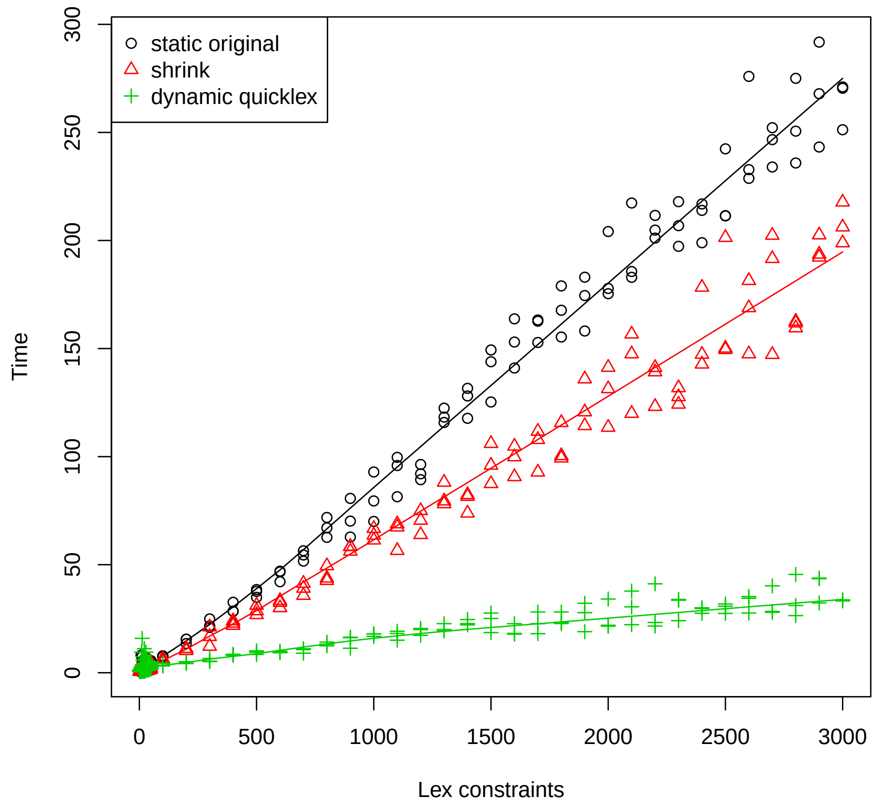

Fig. 3. Comparing the best implementation of lex

2. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.
3. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *CP*, pages 182–197, 2006.
4. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

C-learning: Further Generalised G-nogood Learning

Neil C.A. Moore
ncam@cs.st-andrews.ac.uk

School of Computer Science, University of St Andrews, St Andrews, Scotland, UK.

Abstract. This paper contains practical and theoretical contributions regarding forms of conflict-driven learning more general than g-nogood learning, specifically learning disjunctions composed of arbitrary constraints rather than just assignments and disassignments. A practical implementation of the idea, called c-learning, is given. Most importantly, an exponential separation between c-learning and g-nogood learning is proved: i.e. that is there exists an infinite family of CSPs for which g-nogood learning necessarily takes exponential time, but which c-learning can solve in polynomial time. Finally, to demonstrate the practical possibilities of this result, an experiment is performed demonstrating c-learning's exponential superiority on the family of CSPs used in the proof.

1 Introduction

One possible criticism of state of the art learning in CSP is that though CSP derives its strength from powerful global constraints, CSP learning works on a SAT representation.

The idea of this paper is to investigate how to adapt the g-learning framework to incorporate constraints more general than (dis-)assignments. This is done by means of so-called c-explanations, to be defined rigorously later, but I will give a quick example now to motivate this introduction.

Example 1. The *element* constraint is over a vector of variables V, an index variable i and variable e, and ensures that $V[i] = e$. Suppose that e becomes assigned to 4 and 4 is removed from $\text{dom}(V[7])$. The propagator should detect now that i cannot be set to 7. The best g-explanation for the pruning is $\{e \leftarrow 4, V[7] \not\leftarrow 4\}$.

However if any constraint and not just assignments and disassignments can be used in explanations then another possible explanation is just $\{e \neq V[7]\}$, because whenever e and $V[7]$ are not equal, $i \not\leftarrow 7$. This is a c-explanation because it is composed of constraints rather than simply assignments and disassignments (collectively, (dis-)assignments).

The advantages of c-explanations over g-nogood explanations include:

– They are at least as concise.

- They are more descriptive: in Example 1, the c-explanation covers the inference of $i \not\leftarrow a$ that will result when $e \leftarrow a$ and $V[7] \not\leftarrow a$ for *any* choice of a, rather than just the specific assignment and disassignment that led to the propagation occurring in the current state of search.
- As I will show later, it is often easier to work out a good c-explanation, because the vocabulary available is higher level and often the explanation is recursively related to the definition of the constraint that emits it.
- c-explanations can be less dependent on current domain state. See Example 1 where value 4 is eliminated from the explanation without weakening it.

In the remainder of this paper I will give formal definitions of c-explanations; review the relevant background material; compare their expressiveness versus g-explanations; describe how c-learning can be implemented in an existing g-nogood learning framework; prove that there is an exponential separation between c-learning and g-learning; and demonstrate the separation empirically.

The following is a more rigorous definition of the concept of c-explanation introduced in Example 1.

Definition 1. *A c-explanation for a solver event e*[1] *is a constraint con that is sufficient for the solver to infer e.*

Note 1. It is equally valid to think of a c-explanation as introducing a new reified constraint *con* and reification variable r such that $r \leftrightarrow con$ and then including variable r in the literals of a g-explanation[2].

It should be clear to readers familiar with g-nogood explanations (henceforth g-explanations) [2] that c-explanations are a generalisation of g-explanations (Definition 2).

2 Background

I will now briefly introduce g-nogood learning and other techniques in CSP that generalise it.

2.1 g-nogood learning and lazy explanations

Katsirelos *et al*'s [3, 4, 2] g-nogood learning (g-learning) is a notable CSP search algorithm. In short, whenever the solver reaches a dead-end state, a new constraint is added ruling out other branches that fail for a similar reason.

In order to achieve this, the first step is to analyse the earlier decisions and propagation that contributed to the current failure. The aim is to find a set of assignments and disassignments that, if repeated, lead directly to a failure.

To analyse propagation, *explanations* are used:

[1] e.g. an assignment $x \leftarrow a$
[2] in this respect c-learning incorporates features of extended resolution [1]

Definition 2. *A* g-explanation for disassignment $x \not\leftarrow a$ *is a set of assignments and disassignments that are sufficient for a propagator to infer* $x \not\leftarrow a$*. Similarly a* g-explanation for assignment $y \leftarrow b$ *is a set of (dis-)assignments that are sufficient for a propagator to infer that* $y \leftarrow b$*.*

Example 2. Let a, b and c be three distinct values.

Suppose decision assignments $w \leftarrow a$ and $x \leftarrow a$ have been made. These assignments clearly also cause the remaining values of w and x to be ruled out; we can think of this disassignment being carried out by a built-in "at most one value" constraint. For example now $x \not\leftarrow b$ and the g-explanation for this disassignment is $\{x \leftarrow a\}$.

Now suppose the set of constraints includes both occurrence$([w, x, y, z], b) = 2$ and occurrence$([w, x, y, z], c) = 1$, meaning that variables w, x, y and z must have, respectively, exactly 2 occurrences of b and exactly 1 occurrence of c.

Since $w \leftarrow a$ and $x \leftarrow a$, the former constraint is forced to infer that $y \leftarrow b$ and $z \leftarrow b$. The explanation for both $y \leftarrow b$ and $z \leftarrow b$, for example, is $\{w \not\leftarrow b, x \not\leftarrow b\}$ because when w and x are both not assigned to b, we are forced to set the remainder of the variables to b.

Similarly, since $w \leftarrow a$, $x \leftarrow a$ and $y \leftarrow b$, the latter constraint is forced to infer that $z \leftarrow c$ in order to satisfy the constraint. The g-explanation for $z \leftarrow c$ is $\{w \not\leftarrow c, x \not\leftarrow c, y \not\leftarrow c\}$.

These g-explanations along with the decision assignments can be built into a *implication graph*:

Definition 3. *An* implication graph *for the current state of the variables is a directed acyclic graph where each node is a current (dis-)assignment and there is an edge from* u *to* v *iff* u *appears in the explanation for* v*.*

The g-explanations of Example 2 are displayed as an implication graph in Figure 1.

Repeating the (dis-)assignments in a g-explanation will inevitably lead to the same propagation being repeated, therefore repeating any cut of an implication graph for a failure will inevitably lead to the derivation of the failure again. Hence we build a constraint to avoid that failure by finding a cut $\{c_1, \ldots, c_k\}$ of the implication graph and then adding the constraint to avoid the failure $c = \neg(c_1 \wedge \ldots \wedge c_k)$. Now the solver backtracks and continues.

Example 3. The cut displayed as a dashed line in Figure 1 leads to the constraint $\neg(x \leftarrow a \wedge w \not\leftarrow c \wedge w \not\leftarrow b)$.

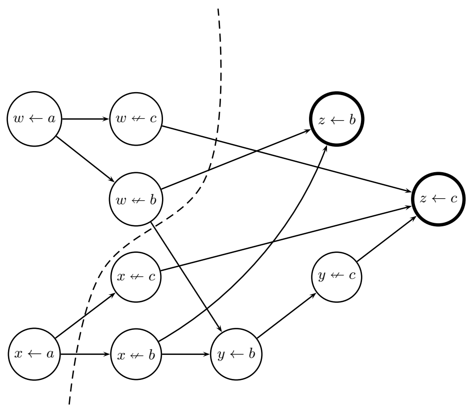

Fig. 1: Implication graph for Example 2. Mutually inconsistent nodes shown with darkened nodes; cut from Example 3 with dashed line.

For correctness and efficiency reasons certain cuts are preferred. In this paper the firstUIP and firstDecision cuts [5] are used, both of which contain exactly one (dis-)assignment that became true at the current depth in search. Both are derived by taking an initial cut, consisting of the (dis-)assignments that directly cause a failure, e.g. $x \leftarrow 1$ and $x \not\leftarrow 1$ Next the algorithm repeatedly replaces the deepest literal by its explanation until a termination condition is reached. For firstUIP the procedure stops when there is exactly one literal from the current decision depth, and for firstDecision it stops when the only literal left is the decision from the current decision depth. After the solver backtracks, the new constraint based on the cut is guaranteed to be unit and will thus propagate resulting in some new progress.

The power of g-learning comes from learned constraints proceeding to propagate and being combined by iterative application of the above process into more powerful constraints that can remove subtrees of the search tree, as opposed to just providing a shortcut to propagation, as in the above examples.

g-nogood learning is extremely effective on some types of benchmark, but has a negative effect on others. Firstly, there is an overhead associated with instrumenting constraint propagators to store explanations, which are needed to produce the new constraints. This problem is mitigated by *lazy learning* [6] which dramatically reduces the overhead of g-learning; however the new constraints must still be propagated and this slows the solver down, too.

2.2 Context

The idea of generalising explanations further than g-learning has appeared several times in the constraints literature.

2.3 Katsirelos' c-nogoods

Katsirelos [3] concludes his thesis by giving a very brief description of various possible techniques that use constraints more general than (dis-)assignments to explain prunings. Katsirelos presents this as the addition of a boolean variable v_C representing the new constraint, i.e. $v_C \leftrightarrow C$ is posted. Now v_C can be incorporated into explanations as appropriate[3].

Katsirelos describes how to use c-nogoods only in the context of logical constraints *and* and *or*. For example, consider the constraint $C_1 \vee C_2$ and suppose that C_1 is disentailed. Using delayed disjunction propagation [7], the remaining disjunct C_2 will be propagated and suppose it causes $v \not\leftarrow a$. A g-explanation for this propagation consists of a g-explanation for the disentailment of C_1, plus a g-explanation for $v \not\leftarrow a$ by C_2. In the c-explanation, all that is needed to explain the entailment of C_1 is the single literal v_{C_1}. Until this paper, no experiments testing this idea have been carried out [3].

Compared to Katsirelos' work, my practical contributions in this paper have been to show how this general idea can be applied to non-logical constraints, to

[3] note the obvious similarity to extended resolution [1]

describe a framework for it to be implemented and to complete an implementation in minion so it can be evaluated empirically. I have also progressed the theoretical understanding of this technique, by proving results about the proof complexity of c-learning versus g-learning.

2.4 Lazy clause generation

Lazy clause generation (LCG) also generalises g-learning by allowing nogoods to contain unary constraints like $x \leq a$ as well as (dis-)assignments. This improves the conciseness of explanations, but not their expressiveness. This is simply because if a clause contains $x \leq a$ is false, for some a, it can easily be replaced by $x \leftarrow a \vee x \leftarrow a-1 \vee x \leftarrow a-2 \ldots$. Hence unlike c-learning, LCG is no more expressive than g-learning.

2.5 Caching using constraints

Learning based on constraints has been tried with some success in the context of *caching* as opposed to constraint learning [8]. Caching is when the search space previously searched is stored as a set of keys, if the current part of search matches a previously searched key then the outcome can be read out of the stored cache. To some extent the distinction between learning and caching is quite artifical: learned constraints are propagated along with the other problem constraints, whereas cached keys are not propagated. Caching relies on keys generalising the subtree in which they are found so that they can be used to avoid search elsewhere. In [8], a "projected key" for each individual constraint is conjoined to form a key for the entire subtree just searched unsuccessfully. For example if the problem contains $c = \text{alldiff}(w, x, y, z)$ s.t. $w, x, y, z \in \{1, 2, 3, 4\}$ and decisions $w = 1$ and $x = 2$ then the projected key for c is $\text{alldiff}(y, z) \wedge y, z \in \{3, 4\}$. This is a key that generalises the subtree from which it is derived, because the constraints in the key are stronger than the problem constraints. The practical results in [8] show that the technique can beat state of the art CSP solvers (with and without learning) on several problem classes.

2.6 Summary

In spite of the approaches described in this section, this paper contains the first practical contribution towards generalising constraint learning beyond unary constraints, as well as fundamental algorithms and theoretical contributions towards understanding the potential of the technique.

3 Expressivity of c-learning

In his thesis, Katsirelos said "there may exist an exponential in the arity of C number of nogoods (g-nogoods) to explain the fact that C is disentailed". This is important because it shows that a single c-nogood is as expressive as an exponential number of g-nogoods. However, it is important that the g-nogoods should also be *minimal*, so that their full power is available. Hence in this section I will prove that a single c-nogood is as expressive as an exponential number of *minimal* g-nogoods.

A strong result can be stated on the relative expressivity of g-explanations and c-explanations. First I must define *prime implicant*:

Definition 4. *An* implicant *I of a boolean formula $f(x)$ is an assignment to a subset of the input arguments of f such that the output of f must be 1. A* prime implicant *is a set minimal implicant, i.e. it can't have assignments removed from it and still be an implicant.*

Prime implicants are related to minimal g-explanations in a simple way:

Lemma 1. A prime implicant of function f is the same as a minimal explanation for $output \leftarrow 1$ in the constraint $output = f(x)$.

Proof. g-explanations must be sufficient for the event they are explaining, and implicants must be sufficient for the output of the circuit to be true. Furthermore, minimal explanations must be setwise minimal, and prime implicants setwise minimal.

For the parity function, there are at least 2^{n-1} different prime implicants:

Fact 1 (given as Proposition 6.1 in [9]) *The parity function $f(x) = (x_1 + \ldots + x_n) \mod 2$ has 2^{n-1} prime implicants of length n each*[4].

Such a set of prime implicants covers each possible input to f whose result is true once and only once, since each implicant includes an assignment to each input. By the correspondence between prime implicants and g-explanations:

Corollary 1. There are 2^{n-1} minimal g-explanations for $output \leftarrow 1$ for constraint $output = \text{parity}(X_1, \ldots, X_n)$.

Proof. By Lemma 1 every implicant is a valid g-explanation. By Fact 1 there are 2^{n-1} distinct prime implicants and hence there are 2^{n-1} distinct minimal g-explanations for $output \leftarrow 1$, one per assignment to $X_1, \ldots, X_n$.

However the c-explanation for $output \leftarrow 1$ in constraint $output = \text{parity}(f)$ is just $\text{parity}(f) = 1$, which is an extremely trivial explanation but exactly captures the required property. Hence when a failure is due to odd parity, 2^{n-1} g-nogoods are required to cover all possible reasons whereas a single c-nogood will do the job. Later, in §5, I will use Corollary 1 to show that entire search trees can be much smaller when c-explanations are used rather than g-explanations. Roughly, this is because with c-nogoods the solver can learn a small powerful constraint like $\text{parity}(f) = 1$ which can cause immediate failure and prove unsatisfiability easily, whereas using g-nogoods it is restricted to enumerating numerous weak constraints until the search space is eventually exhausted.

[4] this is because all prime implicants of parity include assignments to all variables, intuitively because the parity can be changed by flipping a single input

4 Implementing c-learning

Clearly c-explanations generalise g-explanations. They can be substituted into the g-learning framework with only a few changes. However it is necessary to generalise the definition of implication graph (IG) to suit c-learning:

Definition 5 (c-learning implication graph). *An* implication graph *for the current state of variables is a directed acyclic graph where*

- *each node is a currently true constraint, and*
- *there is an edge from u to v iff u appears in the explanation for v.* □

Recall that g-learning requires the following capability for each node in the IG:

- determine at which depth it became entailed (provided by recording the depth of each decision and inference), and
- discover the constraints that are responsible for its entailment (provided by recording an explanation for every propagation).

For the purposes of implementing c-learning, it is usually relatively easy to determine if constraints are entailed: in the worst case each possible assignment could be enumerated in $O(d^a)$ time where d is the domain size and a the arity, and each can be checked for conformance to the constraint in polynomial time. Usually there is a specialised algorithm for each constraint that is efficient.

Since determining entailment is usually easy, so too is discovering the depth at which it became entailed: simply search for the first depth at which it is entailed. However it is better to use tailored algorithms for each constraint where possible.

Discovering the constraints responsible for entailment is done using an explanation procedure, as in g-learning. As an example, I will describe an explanation algorithms for an occurrence constraint propagator in §4.4.

Another thing to notice is that the constraint used to explain the event is *not* necessarily an existing constraint in the CSP, in fact it is quite likely not to be. This is crucially important in practice because it means that the IG *cannot* be built eagerly, while propagation is done, because many of the nodes are brand new constraints. Instead the IG must be uncovered lazily starting with the concrete events that cause failures, as described in [6][5].

Example 4. Following on from Example 1. Suppose that at the current point in time $e \leftarrow 4$ and $v[7] \not\leftarrow 4$, but the propagator for element(V, i, e) has not yet fired. In Example 1, I showed that $\{e \neq V[7]\}$ is a valid c-explanation for the propagation $i \not\leftarrow 7$ that will occur. The constraint $e \neq V[7]$ is in fact entailed by the current domain state, but so are many other constraints[6]. Hence it is infeasible to build a representation of the IG eagerly, because the solver cannot anticipate what constraints will be introduced. Once the propagation $i \not\leftarrow 7$ has occurred, the constraint $e \neq V[7]$ becomes concrete.

[5] in g-learning being lazy is useful but not essential, here it is essential

[6] e.g. any constraint satisfied by any remaining assignment to any possible subset of the current variables

Conversely, in g-learning, the constraints that can become involved in the IG are known at all times: it's just the set of current assignments and disassignments.

4.1 Required properties of c-explanations

c-explanations being used in IGs and processed to find a firstUIP cut using the procedure alluded to in §2.1 must conform to certain properties. Suppose explanation $\{c\}$ labels event e:

Property 1. The entailment depth of c may not be greater than the depth of event e.

Remark 1. Ensures that causes precede effects, ensuring no cycles in the implication graph.

Property 2. Paths in the IG must be finite, i.e. c-explanations must eventually bottom out to (dis-)assignments.

Remark 2. This property is implicit in g-learning, for since the edges always go from nodes with a higher to a lower decision depth, paths must be finite. In c-learning this is not automatically the case, because it would be possible for an infinite path of virtual constraints to occur with the same entailment, e.g. two equivalent constraints that each explain their own entailment using the other. An infinite path might mean a cut cannot be computed in a finite number of steps.

4.2 Propagating clauses consisting of arbitrary constraints

One of the fundamental ingredients that makes nogood learning work is that the clauses learned are guaranteed to propagate on backtrack, so that progress is always made. Suppose that firstUIP cut $\{c_1, \ldots, c_k\}$ is added as nogood $(\neg c_1 \vee \ldots \vee \neg c_{k-1} \vee \neg c_k)$. By the properties of firstUIP, $c_1, \ldots, c_{k-1}$ are all disentailed when the constraint is posted. Hence c_k will be unit propagated.

My solver uses watched literals to propagate arbitrary disjunctions of constraints (*watched or*) [10]. Using watched or, each disjunct constraint must be implemented with a *complete satisfying set generator*, which means that the watched or propagator can detect as soon as it has become disentailed (see [10]). This means that unit propagation can happen as soon as possible.

In the case of g-learning, c_k is guaranteed to propagate, since a (dis-)assignment can always propagate successfully. However I will now show that this is not the case in c-learning, by exhibiting a counterexample:

Example 5. Consider the CSP consisting of variables $V[1], V[2], X$ and Y each with domain $\{0, 1\}$ and constraints

- occurrence$(V, 1) \leq 1 \leftrightarrow X$,
- occurrence$(V, 1) \leq 1 \leftrightarrow Y$, and

– $X \vee Y$.

Suppose that $V[1] \leftarrow 1$ at depth 1.0. No propagation is possible by any constraint (this is less obvious for the bi-implications than for $X \vee Y$, but it can be verified by inspecting all possible assignments over the scope $V[1], V[2], X$ (similarly for Y) which consist of

$$\{(0,0,1),(0,1,1),(1,0,1),(1,1,0)\}$$

for both bi-implications.

Suppose next that $V[2] \leftarrow 1$ at depth 2.0. Now the left hand size of each bi-implication constraint is definitely disentailed, and the bi-implications will propagate $X \leftarrow 0$ and $Y \leftarrow 0$ respectively. Hence clause $(X \vee Y)$ is empty and conflict analysis will follow. The implication graph is shown in Figure 2. Clearly occurrence$(V,1) > 1$ is a c-explanation for assignments $X \leftarrow 0$ and $Y \leftarrow 0$. The firstUIP cut is actually just $\{\text{occurrence}(V,1) > 1\}$. Conflict analysis will therefore backjump to depth 0 and attempt to propagate the constraint occurrence$(V,1) \leq 1$. The occurrence constraint cannot rule out any value and so no propagation will occur, as I set out to show.

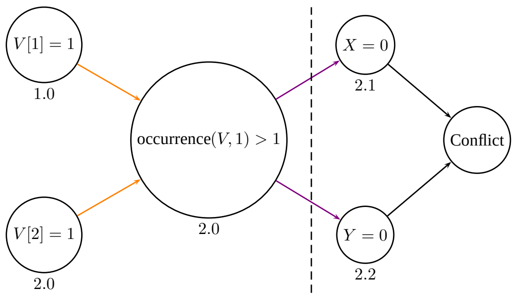

Fig. 2: Implication graph for Example 5

However the firstDecision cut is guaranteed to propagate because the disjunct that is unit will definitely be a (dis-)assignment. Hence the approach taken in c-learning is first to try the firstUIP constraint, monitoring if any propagation occurs, and if not, revoke it and add the firstDecision cut, which is guaranteed to result in some progress. Hopefully this will not be necessary very often, but it is essential for correctness. Note that the benefits of c-learning do not require that any additional propagation occurs immediately after backtrack. In fact, the more generalised disjuncts need never themselves become unit: they need only be violated more often and thus help other clauses to become unit more often than in the case of g-learning.

4.3 Common subexpression elimination

Common subexpression elimination is when the same constraint expression posted twice is replaced by a single occurrence of the expression. For example, consider the following example from [11]. Expression $a + x \times y = b \wedge b + x \times y = t$ might typically be flattened to $aux_1 = x \times y \wedge a + aux_1 = b \wedge aux_2 = x * y \wedge b + aux_2 = t$. However when common subexpressions are taken into account, $aux_1 = x \times y \wedge a+$

$aux_1 = b \land b + aux_1 = t$ is a smaller and more strongly propagating alternative [11]. See [12] for more information.

For logical constraints like disjunction, there can be an advantage to recognising common disjuncts. The reason for this, is that, in general, there is a difference between a constraint being forced to be satisfied, and being currently entailed. For example, suppose that C_2 is enforced by unit propagation on $C_1 \lor C_2$. Although C_2 is forced to be satisfied, it is not necessarily entailed, so constraint $\neg C_2 \lor C_3$ may not become unit. Hence, disjunction propagation should be implemented to unit propagate when all but one disjunct is *either* entailed or forced to be true. I have implemented this feature in my solver for the special case described in §6.1.

4.4 Explainers

As with g-learning, much of the effort in implementing c-learning is providing small and correct explanations for each (dis-)assignment caused by a propagator. Please note that the following c-explanations are not implemented, and hence no experiments are included to compare them with the corresponding g-explanations.

Occurrence The constraint $occurrence(V, i) \leq count$ ensures that there are at most *count* occurrences of value i in vector V. In minion, i is a constant but both V and *count* are variables. The constraint $occurrence(V, i) \geq count$ is also available in minion, however I will only describe how to derive explanations for *occurrence* $\leq$, since *occurrence* $\geq$ is symmetric.

The minion propagator for *occurrence* $\leq$ propagates in the following cases:

- when i is already assigned max(dom(*count*)) times, the constraint would be failed if any more were assigned, so i is removed from all the other domains; and
- remove any values from dom(*count*) that are smaller than the current number of assignments in V to value i.

Note that both the g- and c-explanations for *occurrence* $\leq$ described in this section are original to this paper.

Explanation for $V[idx] \nleftarrow i$ The c-explanation for this type of propagation is very simple. Suppose that $V[idx] \nleftarrow i$ by the first propagation rule above. It must be that the number of occurrences of i in V *excluding* position idx is already max(*count*). Hence the explanation is simply $occurrence(V[1, \ldots, idx - 1, idx + 1, \ldots, |V|], i) \geq count$.

A minimal g-explanation is the set of max(dom(*count*)) assignments of variables in V to value i, unioned with the set of prunings to *count* above max(dom(*count*)).

The c-explanation generalises the g-explanation in a number of ways:

1. If a different set of assignments makes the total number of i's greater than $\max(\text{dom}(\mathit{count}))$, the explanation will still apply, since it does not specify which variables in V are assigned.
2. If $\max(\text{dom}(\mathit{count}))$ is smaller or larger elsewhere in search and the number of i's again reaches $\max(\text{dom}(\mathit{count}))$, the explanation will still be valid.

I will now show how many different minimal g-explanations each c-explanation covers. In the following, I assume that the domain of *count* is entirely non-negative, for any negative numbers would be pruned out immediately anyway. The c-explanation $\mathit{occurrence}(V[1,\ldots,\mathit{idx}-1,\mathit{idx}+1,\ldots,|V|],i) \geq \mathit{count}$ covers

$$\sum_{j=\min(\text{dom}(\mathit{count}))}^{\max(\text{dom}(\mathit{count}))} \binom{|V|}{j} = 2^{|\text{dom}(\mathit{count})|}$$

because for each possible value for $\max(\text{dom}(\mathit{count}))$, any set of that many assignments of variables in V to value i can be chosen. As shown, this sum is exponential in *count* [13].

Explanation for $\boldsymbol{\mathit{count} \not\leftarrow c}$ Suppose that a propagator for $\mathit{occurrence} \leq$ has caused $\mathit{count} \not\leftarrow c$. The c-explanation is $\mathit{occurrence}(V,i) \geq c+1$. This is because by the second propagation rule above, $c \in \text{dom}(\mathit{count})$ is pruned when the count of i's exceeds c.

A minimal g-explanation is the set of assignments of variables in v to value i.

The c-explanation generalises the g-explanation because it captures any possible set of at least $c+1$ assignments to V.

Each c-explanation captures exactly $\binom{|V|}{c+1}$ g-explanations, that is, all the ways to set $c+1$ variables in V to i.

5 Proof complexity of c-learning

I will now prove that c-learning can be significantly superior to g-learning: there is an exponential separation between the two, meaning that there exists an infinite family of instances of increasing size parameter n such that any backtracking search algorithm using g-nogood learning takes at least exponential time in n using any possible search strategy whereas there is a simple algorithm that learns c-nogoods that can solve any such problem in time polynomial in n. First some definitions are required:

Definition 6. *The constraint $\mathit{parity}(X)$ ensures that $(\sum_i X_i) \equiv 1 \bmod 2$, where X is a boolean vector. Hence $\neg\mathit{parity}(X)$ is just $(\sum_i X_i) \equiv 0 \bmod 2$.*

This constraint is interesting for several reasons. The first is that until all but one of the variables is instantiated, a propagator cannot prune any values:

Lemma 2. *No propagator for parity(X) can remove any values until $|X| - 1$ variables are instantiated.*

Proof. Let I be the proper subset $I \subset X$ of size k that are instantiated. Suppose $|X \setminus I| > 2$, i.e. fewer than $|X| - 1$ variables are instantiated. Let $x \in X \setminus I$ be arbitrary and let $others = X \setminus I \setminus \{x\}$. Suppose that the sum of I is either congruent to 1 (resp. 0) modulo 2. Then $0 \in \mathrm{dom}(x)$ is supported because *others* can be assigned s.t. $\sum others \equiv 0 \bmod 2$ (resp. $\sum others \equiv 1 \bmod 2$). Also $1 \in \mathrm{dom}(x)$ is supported because *others* can be assigned s.t. $\sum others \equiv 1 \bmod 2$ (resp. $\sum others \equiv 0 \bmod 2$). Hence 0 and 1 are supported for all uninstantiated variables if $|X \setminus I| > 2$, as required.

The second required fact is that *parity*(X) is not entailed until all $|X|$ variables are instantiated. This should be obvious from the previous lemma and its proof.

Lemma 3. *parity(X) is not entailed until $|X|$ variables are instantiated.*

I can now introduce the infinite family of problems of increasing size used to prove the result, parameterised by n:

Definition 7. *CSP $M(n)$ consists of variable x and vector of variables X of length n, each of which has a $\{0, 1\}$ domain, and constraints*

$$x \leftarrow 1 \vee parity(X) \tag{1}$$
$$x \leftarrow 1 \vee \neg parity(X) \tag{2}$$
$$x \not\leftarrow 1 \vee parity(X) \tag{3}$$
$$x \not\leftarrow 1 \vee \neg parity(X) \tag{4}$$

There are various techniques that would make this instance very easy, such as remodelling the problem by reifying *parity*(X), and I seek to prove that c-learning is one such technique. The proof relies on the fact that it should be possible to discover when $r \leftrightarrow C$ or $r \leftrightarrow \neg C$ has already been introduced by the learning process, and to reuse r in future explanations where possible. In practice this facility will save memory and also can be used to improve propagation (see §4.3). It is also necessary to prove that g-learning will necessarily find $M(n)$ hard no matter how clever it is. First I will prove that c-learning will find it easy to show that there are no solutions to $M(n)$ for any n:

Lemma 4. *For any given n, c-learning can prove $M(n)$ unsatisfiable in polynomial time.*

Proof. Assign the variables in vector X so that *parity*(X) is entailed, e.g. assignment $1, 0, 0, 0, \ldots$, then disjunctions 2 and 4 can unit propagate to cause $x \leftarrow 1$ and $x \not\leftarrow 1$. Hence $\{\neg parity(X)\}$ is the firstUIP cut for this conflict. Constraint $r \leftrightarrow \neg parity(X)$ will be introduced, where r is a fresh variable, and the constraint $r \leftarrow 1$ learned.

Next assign vector X so that $\neg parity(X)$ is entailed, then similarly to the above $\{parity(X)\}$ is the firstUIP cut. The constraint learned is $r \leftarrow 0$, since $r \leftrightarrow \neg parity(X)$ was introduced earlier.

A conflict at the root node is guaranteed because r is forced to be both 0 and 1.

Clearly this can be implemented in polynomial time for any n.

Finally I will prove that $G(n)$ is necessarily hard for g-learning, even when arbitrary variable and value ordering is allowed:

Lemma 5. *For any given n, g-learning takes exponential time to prove $M(n)$ unsatisfiable using any variable ordering.*

Proof. Suppose that every variable in X is assigned before x. Then w.l.o.g. and by Lemma 3, $parity(X)$ (or $\neg parity(X)$) is entailed as soon as the last assignment is made and not before. Hence disjunctions 2 and 4 will propagate to force a conflict in variable x. The conflict analysis process must include every assignment to X, since by Lemma 3 all are required to ensure entailment of $parity(X)$ (or $\neg parity(X)$), without which the conflict cannot occur. This nogood rules out only the current assignment.

The case where x is assigned before X is fully assigned is only slightly more complex. By Lemma 2, until all but one variable x_u in X is assigned, there is no chance of any propagation. Suppose w.l.o.g. that $x \leftarrow 1$ ($x \leftarrow 0$) when this happens. Now disjunctions 1 and 2 will unit propagate to force the remaining variable x_u to be both 0 and 1, which is required to satisfy unit implicants $parity(X)$ and $\neg parity(X)$ respectively. Hence a conflict results. The g-nogood must involve $x \leftarrow 1$ without which 1 and 2 cannot unit propagate and the entire assignment to X apart from x_u without which the parity constraints cannot propagate. This rules out only the current assignment.

Since by any possible variable and value ordering, each g-nogood only rules out one partial assignment complete except for one variable, 2^n partial assignments must be tried before the search space is exhausted and hence the algorithm takes exponential time.

The previous lemmas combine in the obvious way to give:

Theorem 1. *There is an exponential separation between g-learning and c-learning.*

Recall that Theorem 4 takes advantage of common subexpression detection, it is an open question whether the Theorem can be proved without it. This proof does not allow for restarts during search. There is no reason to believe the result does not hold in the presence of restarts, but I have not proved it rigorously.

6 Experimental results

To see this in practice, I have implemented the parity constraint and have tried the above problem in my c-learning solver. In addition I provide results on antichain problems.

n	c-learn time	c-learn nodes			g-learn time	g-learn nodes		
	Mean (secs)	Min	Mean	Max	Mean (secs)	Min	Mean	Max
01	0.006	1	1	1	0.006	1	1	1
02	0.006	2	2	2	0.006	3	3	3
03	0.006	3	3	3	0.006	7	7	7
04	0.006	4	4	4	0.007	15	15	15
05	0.006	5	5	5	0.007	31	31	31
06	0.006	6	6	6	0.009	63	63	63
07	0.006	7	7	7	0.013	127	127	127
08	0.007	8	8	8	0.021	255	255	255
09	0.007	9	9	9	0.041	511	511	511
10	0.007	10	10	10	0.093	1023	1023	1023
11	0.007	11	11	11	0.226	2047	2047	2047
12	0.007	12	12	12	0.589	4095	4095	4095
13	0.007	13	13	13	1.723	8191	8191	8191
14	0.007	14	14	14	5.399	16383	16383	16383
15	0.007	15	15	15	28.726	32767	32767	32767
16	0.007	16	16	16	34.970	65535	65535	65535
17	0.007	17	17	17	47.563	131071	131071	131071
18	0.007	18	18	18	117.050	262143	262143	262143
19	0.007	19	19	19	279.564	524287	524287	524287

Table 1: Comparison of c- and g-learning on parity instances

6.1 Results on family $M(n)$

$M(n)$ is the problem used in §5 for proof complexity results for c-learning.

Procedure I have run $M(n)$ for $n = 1$ to 19. The possibility of fast execution for c-learning is proved by running it according to the variable and value ordering described in Lemma 4. In order to demonstrate empirically that g-learning is slow I have run instances up to 19 variables 100 times each using a random variable ordering.

Implementation The g-learning solver uses lazy explanations [6]; does not forget constraints or restart; and learns the firstUIP cut. The explainer for parity is unique to this paper and uses minimal explanations.

The c-learning solver is based on the same solver, but uses a different explainer for watched OR [10]. Specifically, when a watched OR $C \vee D$ propagates D because C is disentailed, the explanation is $\neg C \cup E$ where E is the explanation for D's propagation. In order to detect when C or $\neg C$ is reintroduced by the learning system, each new constraint is added to a list when it is first posted. If the negative of an existing constraint is posted, search is stopped. This implementation is not very good and not as powerful as common subexpression detection, but does give polynomial performance and suffices for present purposes.

Results Table 1 demonstrates convincingly that c-learning is much better at $M(n)$ than g-learning. c-learning solves the problem in the same number of nodes as there are variables. g-learning takes $2^n - 1$ nodes as predicted by the proof of Lemma 5. It is worth pointing out that no matter what the ordering used, this number does not change, again as predicted by the lemma's proof.

Discussion I appreciate that this problem is artificial in nature and arises only as a means of proving results in proof complexity. Hence in the following section I reproduce experiments on parity problems.

6.2 Antichain experiments

This section describes experiments applying c-learning to CSPs modelling antichains. First I will define what an antichain is and then describe the CSP used to find them.

An *anti-chain* is a set S of multisets where $\forall \{x, y\} \subseteq S.\ x \not\subseteq y \wedge y \not\subseteq x$. In other words, the $< n, l, d >$ anti-chain is a set of n multisets with cardinality l drawn from d elements in total, such that no multiset is a subset of another.

In [10] this is modelled as a CSP using n arrays of variables, denoted $M_1, \ldots, M_n$, each containing l variables with domain $\{0, \ldots, d-1\}$ and the constraints $\forall i \neq j \in \{1, \ldots, n\}.\ \exists k \in \{1, \ldots, n\}.\ M_i[k] < M_j[k]$. Each variable $M_i[v]$ represents the number of occurrences of value v in multiset i, up to a maximum of $d-1$. Each pair of rows M_i and M_j differ in at least two places: in one position k, $M_i[k] < M_j[k]$ and in another position p, $M_i[p] > M_j[p]$. This ensures that neither multiset contains the other. The constraint $\exists i.\ M[i] < N[i]$ for arrays M and N is encoded as a watched or as follows: $M[0] < N[0] \vee \ldots \vee M[l] < N[l]$.

This problem appears quite suitable for evaluating c-learning because the watched or explanation (see §2.3) introduces many $<$ constraints into the implication graph. Furthermore, it is relatively easy to detect when a $<$ constraint is entailed or disentailed, so the learned constraints should be relatively efficient to propagate.

Experimental methodology Each of the antichain instances was executed five times with a 10 minute timeout, over 4 Linux machines each with 2 Intel Xeon cores at 2.4 GHz and 2GB of memory, running kernel version 2.6.18 SMP. Parameters to each run were identical, and the minimum time for each is used in the analysis, in order to approximate the run time in perfect conditions (i.e. with no system noise) as closely as possible. Each instance was run on its own core, each with 1GB of memory. Minion was compiled statically (`-static`) using g++ version 4.4.3 with flag `-O3`.

The g-learning solver used is the same as described in §6.1. Two different variable orderings are used and reported separately: lexicographical and dom/wdeg.

Results Table 2 shows the time and nodes taken to solve a selection of antichain instances. The instances were chosen to include a range of different search sizes and problem sizes. These results show that, for these instances, c-learning is not able to significantly reduce the space searched. Hence, the CPU time is also worse for c-learning, as expected, because the overhead of adding generalised constraints and maintaining the c- implication graph is greater. A speedup would only result due to a large decrease in nodes.

Instance	**Lex ordering**				**domoverwdeg**			
	C nodes	C time	G nodes	G time	C nodes	C time	G nodes	G time
<2,2,2>	2	0.21	2	0.21	2	0.21	2	0.21
<6,4,4>	16	0.21	16	0.21	16	0.22	16	0.21
<7,3,3>	832	2.00	809	0.51	637	1.30	686	0.53
<8,3,3>	???	Time out.	14150	22.75	???	Time out.	23817	357.87
<8,3,8>	1506	45.15	1529	2.90	56	0.23	61	0.24
<8,4,5>	346	1.03	350	0.42	327	0.94	297	0.47

Table 2: Comparison of strategies for solving antichain

Instance	Median clause length (G)	Median clause length (C)	%C UIPS	C%
6-4-4	19.0	10.0	1.00	0.90
7-3-3	14.0	18.0	0.78	0.95
8-3-3	17.0	26.0	0.88	0.94
8-3-8	80.0	28.0	0.31	0.74
8-4-5	59.0	51.0	0.70	0.82

Table 3: Runtime statistics for antichain instances using wdeg ordering

Instance	Median clause length (G)	Median clause length (C)	%C UIPS	C%
6-4-4	29.0	25.0	0.60	0.76
7-3-3	16.0	24.0	0.85	0.83
8-3-3	20.0	30.0	0.80	0.89
8-3-8	80.0	63.0	0.67	0.68
8-4-5	57.0	52.5	0.72	0.85

Table 4: Runtime statistics for antichain instances using lex ordering

I will now supply some further runtime statistics on both solver types, in Tables 3 and 4. The former table gives statistics for dom/wdeg variable ordering and the latter for lexicographical ordering. The columns are as follows: *Median clause length* – The median number of disjunctions in learned constraints, *%C UIPS* – The percentage of the time that a non (dis-)assignment is the UIP. *C%* The median over all constraints of percentage of disjuncts that are not (dis-)assignments.

There is no apparent problem with the results for the latter two statistics. They show that most of the time, the UIP is a constraint rather than a (dis-)assignment, allowing for the possibility of stronger propagation. They also show that the clauses are made up primarily of constraints, allowing for better inference. The clause length statistics are more problematic, because the difference between g- and c-learning lengths is usually relatively small, although one would hope the c-learning constraints would be shorter since they are more expressive.

Discussion I do not know why c-learning does not work for the antichain instances. I believe that good c-learning constraints should be significantly shorter than g-learning constraints, since they are more expressive. Extrapolating from the parity experiments in §6.1, c-learning appears to be powerful when long g-learning constraints can be replaced by short c-learning constraints. The fact that in these experiments, constraint length is similar is a cause for concern. I imagine that a different method for deriving cuts may be useful to achieve this, for example one that minimises cut width.

In conclusion, more needs to be done to see if the promise of the experiments in §6.1 extends to problems of practical interest. I leave large scale evaluation of more problems and constraint types to future work.

7 Conclusions and discussion

In this paper I have made practical and theoretical contributions to the understanding of c-learning. First I described the idea and how to implement it in a practical solver. I also described how to produce c-explanations for the occurrence constraint precisely quantifying the difference in expressivity between g- and c-explanations. Next, I answered an open question from the "Future work" section of [3]. The proof showed that g-learning requires exponentially more search to solve a family of CSPs compared to c-learning. It used a new approach that does not rely on previous work on proof complexity in SAT, unlike many proofs of this type in the past. To demonstrate the practical possibilities of this result, I performed an experiment showing c-learning's exponential superiority over g-learning on the family of CSPs used in the proof.

References

1. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic **Part II** (1968) 115–125
2. Katsirelos, G., Bacchus, F.: Generalized nogoods in csps. In Veloso, M.M., Kambhampati, S., eds.: AAAI, AAAI Press / The MIT Press (2005) 390–396
3. Katsirelos, G.: Nogood Processing in CSPs. PhD thesis, University of Toronto (Jan 2009) `http://hdl.handle.net/1807/16737`.
4. Katsirelos, G., Bacchus, F.: Unrestricted nogood recording in CSP search. In: CP. (2003) 873–877
5. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285
6. Gent, I., Miguel, I., Moore, N.: Lazy explanations for constraint propagators. In: PADL 2010. Number 5937 in LNCS (January 2010)
7. Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(fd). J. Log. Program. **37**(1-3) (1998) 139–164
8. Chu, G., de la Banda, M.G., Stuckey, P.J.: Automatically exploiting subproblem equivalence in constraint programming. In Lodi, A., Milano, M., Toth, P., eds.: CPAIOR. Volume 6140 of LNCS., Springer (2010) 71–86
9. Wegener, I.: The complexity of Boolean functions. Wiley-Teubner (1987)
10. Jefferson, C., Moore, N.C., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artificial Intelligence Journal (AIJ) **174** (November 2010) 1407–1420
11. Rendl, A., Miguel, I., Gent, I.P., Jefferson, C.: Automatically enhancing constraint model instances during tailoring. In: SARA, AAAI (2009)
12. Rendl, A.: Effective Compilation of Constraint Models. PhD thesis, School of Computer Science, University of St Andrews (2010)
13. Rosen, K.H.: Discrete mathematics and its applications (2nd ed.). McGraw-Hill, Inc., New York, NY, USA (1991)

Between Path-Consistency and Higher Order Consistencies in Boolean Satisfiability

Pavel Surynek[*]

Charles University in Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. The task of enforcing certain level of consistency in Boolean satisfiability problem (SAT problem) is addressed in this paper. A recently developed concept of modified variant of path-consistency (PC) is recalled and experimentally evaluated. The modified PC combines the standard PC on the literal encoding of the given SAT instance with global properties calculated from constraints imposed by the instance – namely with the maximum number of visits of a certain set by the path being checked to exist. An experimental evaluation discovered that modified PC performs well on difficult SAT instances with structured constraint graph. However, the performance was not so good on other classes of SAT instances. Therefore the possible limits of improvements of modified PC were investigated by evaluating (2,*k*)-consistency which represent the strongest possible variant of modified PC in the given context. This evaluation showed that not only there are SAT instances for which it worth to study improvements of modified PC (namely they are represented by the difficult integer factorization problems) but also that (2,*k*)-consistency consistency itself can be used as an efficient preprocessing tool.

Keywords: local consistency, global consistency, path-consistency, (2,*k*)-consistency, CSP, SAT

1 Introduction and Motivation

A method for increasing the inference strength of *path-consistency* (PC) [14, 15] with regard on applications in Boolean satisfiability (SAT) [6] called *modified PC* [20] is revisited in this paper. The work on modified PC is still in progress and this paper should be regarded as a next step in this progress.

The concept of modified PC combines the standard PC on the literal encoding model [21] of the SAT instance [6] with global properties calculated from constraints forming the instance. The existence of a path in a graph interpretation of the instance

[*] This work is supported by The Czech Science Foundation (Grantová agentura České republiky - GAČR) under the contract number 201/09/P318 and by The Ministry of Education, Youth and Sports, Czech Republic (Ministerstvo školství, mládeže a tělovýchovy ČR – MŠMT ČR) under the contract number MSM 0021620838.

which is normally being checked within PC is further restricted by additional requirements. Regarding the inference strength, modified PC lies between $(2,k)$-consistency [8] and standard PC (see Fig. 1 for illustration). The goal is to tune modified PC so that it will eventually represents a good trade-off between PC and $(2,k)$-consistency in terms of strength and computational cost. The ultimate goal of whole design of modified PC is a tool for preprocessing SAT instances.

In this paper, we investigate positive and negative properties of the current version of modified PC and assess its possible improvements by evaluating $(2,k)$-consistency. Surprisingly, this evaluation showed that $(2,k)$-consistency itself can be successfully used as a SAT preprocessing tool. The paper has two parts; in the first part modified PC is recalled; in the second part experimental evaluation is presented and its implications are discussed.

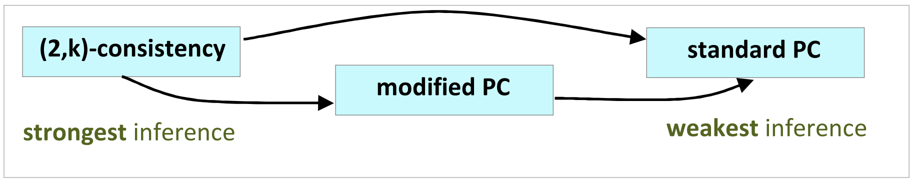

Fig. 1. *The concept modified PC in the context of other consistencies.* The inference strength is depicted using arrows. The goal is to shift modified PC towards (2,*k*)-consistency.

2 Notations and Definitions

Concepts of *constraint satisfaction problem* (CSP) [8] and *Boolean satisfiability* (SAT) [6] need to be established first to make reasoning about PC in the context of SAT easier to understand.

Definition 1 (*Constraint satisfaction problem - CSP*). Let $\mathbb{D}$ be a finite set representing *domain universe*. A *constraint satisfaction problem* [8] is a triple (X,C,D) where X is a finite set of *variables*, C is a finite set of *constraints*, and $D: X \longrightarrow \mathcal{P}(\mathbb{D})$ is a function that defines *domains* of individual variables from X (that is, $D(x) \subseteq \mathbb{D}$ is a set of values that can be assigned to the variable $x \in X$). Each constraint from $c \in C$ is of the form $\langle (x_1^c, x_2^c, \dots, x_{K^c}^c), R^c \rangle$ where $K^c \in \mathbb{N}$ is called an arity of the constraint c, the tuple $(x_1^c, x_2^c, \dots, x_{K^c}^c)$ with $x_i^c \in X$ for $i = 1,2,\dots,K^c$ is called a *scope* of the constraint, and the relation $R^c \subseteq D(x_1^c) \times D(x_2^c) \times \dots \times D(x_{K^c}^c)$ defines the set of tuples of values for that the constraint c is satisfied. The task is to find a valuation of variables $v: X \longrightarrow \mathbb{D}$ such that $v(x) \in D(x)$ $\forall x \in X$ and $(v(x_1^c), v(x_2^c), \dots, v(x_{K^c}^c)) \in R^c$ $\forall c \in C$. □

A constraint $c \in C$ with the scope $(x_1^c, x_2^c, \dots, x_{K^c}^c)$ will be denoted as $c(\{x_1^c, x_2^c, \dots, x_{K^c}^c\})$; this notation is useful when the ordering of variables in the scope is not known from the context; when ordering of variables in the scope matters, then a notation $c(x_1^c, x_2^c, \dots, x_{K^c}^c)$ will be used instead.

A CSP is called *binary* if all the constraints has the arity of two. The expressive power of a binary CSP is not reduced in comparison with a general one since every

CSP can be transformed into an equivalent binary CSP [17]. The key concept of *path-consistency* (PC) [15] that is addressed in this paper is defined for binary CSPs only. It is also convenient to suppose, that each pair of variables is constrained by at most one constraint.

Definition 2 (*Boolean satisfiability problem - SAT*). Let B be a finite set of *Boolean variables*; that is, a set of variables that can be assigned either $FALSE$ or $TRUE$. A Boolean formula F over the set of variables B in a so called *conjunctive normal form* (*CNF*) [13] is the construct of the form $\bigwedge_{i=1}^{N}(\bigvee_{j=1}^{K_i} l_j^i)$ where l_j^i with either $l_j^i = y$ or $l_j^i = \neg y$ for some $y \in B$ for $i = 1,2,\dots,N$; $j = 1,2,\dots,K_i$ is called a *literal* and $(\bigvee_{j=1}^{K_i} l_j^i)$ for $i = 1,2,\dots,N$ is called a *clause*. The task is to find a valuation of Boolean variables $b: B \longrightarrow \{FALSE, TRUE\}$ such that F evaluates to $TRUE$ under b while $\neg$ (*negation*), $\vee$ (*disjunction*), and $\wedge$ (*conjunction*) are interpreted commonly in the Boolean algebra. A formula for that such a satisfying valuation exists is called *satisfiable*. □

2 Path-consistency in CSP

The standard definition of *path-consistency* in CSP will be recalled before the augmented versions and their relaxations are introduced. The following definition refers to general paths of variables which is not necessary in fact. However, this style of definition will be more suitable for making intended augmentations.

Definition 3 (*Path-consistency - PC*). Let (X, C, D) be a binary CSP and let $P = (x_0, x_1, \dots, x_K)$ with $x_i \in X$ for $i = 0,1,\dots,K$ be a sequence of variables called a *path*. A pair of values $d_0 \in D(x_0)$ and $d_K \in D(x_K)$ is *path-consistent* with respect to P if there exists a valuation $v: \{x_0, x_1, \dots, x_K\} \longrightarrow \mathbb{D}$ with $v(x_0) = d_0 \wedge v(x_K) = d_K$ such that constraints $c(\{x_i, x_{(i+1) \bmod K}\})$ are satisfied by v for every $i = 0,1,\dots,K$. The path P is said to be *path-consistent* if all the pairs of values from $D(x_0)$ and $D(x_K)$ respectively are path-consistent with respect to P. Finally, the CSP (X, C, D) is said to be *path-consistent* if it is path-consistent for every path. □

Notice that variables forming the path in the definition do not need to be necessarily distinct. Although the notion of PC seems to be computationally infeasible since there are typically too many paths, it is sufficient to check PC for all the paths consisting of triples of variables only to ensure that the given CSP is path-consistent [14, 15]. In other words, although it seems that PC captures the problem globally (a path can go through large portion of variables of the instance), it merely defines a local property.

There exist many algorithms for enforcing PC in a CSP such as PC-4 [11] and PC-6 [1, 4]. They differ in the representation of auxiliary data structures and the efficiency. The common feature of PC algorithms is however the process how the consistency is enforced. It is done by eliminating pairs of inconsistent values until a path-consistent state is reached (the smallest set of pairs of values such that their elimination makes the problem path-consistent is being pursued). The process of elimination of pairs of values is typically done by pruning extensional representation of constraints (lists of allowed tuples) to forbid more pairs of values.

3 Standard Path-Consistency in SAT

The aim of this work is to modify PC to make it applicable on SAT and to increase its inference strength by incorporating certain global reasoning into it. The easier task is to make PC applicable on SAT - it is sufficient to model SAT as CSP. A so called *literal encoding* [21], which of the result is a binary CSP, is particularly used. This kind of encoding is especially suitable since it allows natural expressing of PC in terms of graph constructs.

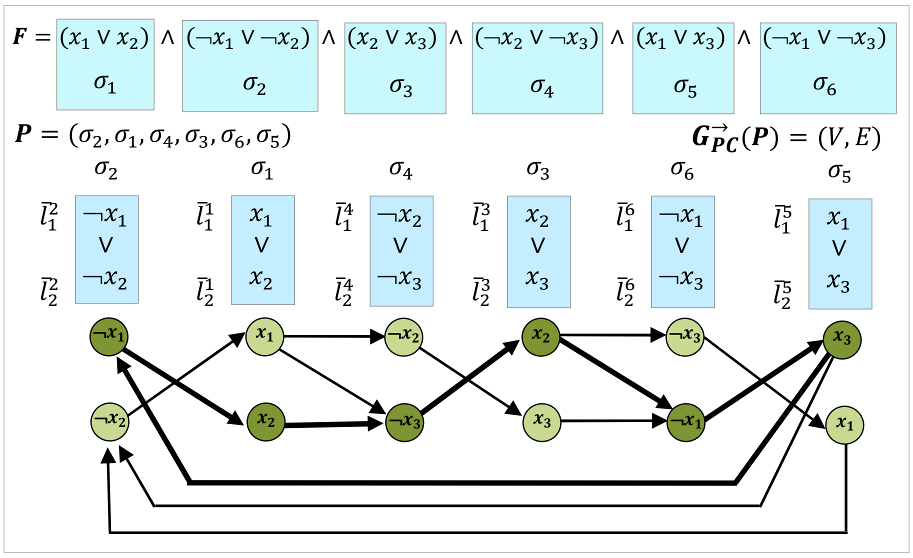
$F = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$
σ_1 σ_2 σ_3 σ_4 σ_5 σ_6
$P = (\sigma_2, \sigma_1, \sigma_4, \sigma_3, \sigma_6, \sigma_5)$
$G_{\overrightarrow{PC}}(P) = (V, E)$
$\bar{l}_1^2$ $\bar{l}_2^2$ $\bar{l}_1^1$ $\bar{l}_2^1$ $\bar{l}_1^4$ $\bar{l}_2^4$ $\bar{l}_1^3$ $\bar{l}_2^3$ $\bar{l}_1^6$ $\bar{l}_2^6$ $\bar{l}_1^5$ $\bar{l}_2^5$
$\neg x_1$ x_1 $\neg x_2$ x_2 $\neg x_1$ x_3
$\neg x_2$ x_2 $\neg x_3$ x_3 $\neg x_3$ x_1

Fig. 2. *An illustration of path-consistency in the CSP model of a SAT problem.* The SAT problem represented by a formula F shown here is a representation of the requirement of selecting an odd number of variables from every of the following sets to be true: $\{x_1, x_2\}$, $\{x_1, x_3\}$, $\{x_2, x_3\}$. Observe, that there is no satisfying valuation of F. However, a pair of literals $\neg x_1$ and x_3 from the left most variable and from the right most variable respectively are path-consistent with respect to a depicted path P since they are non-conflicting and there exists a path from the left to the right consisting of edges between neighboring variables connecting allowed pairs of values (the path is marked by bold edges and by darker vertices).

Let $F = \bigwedge_{i=1}^{N}(\bigvee_{j=1}^{K_i} l_j^i)$ be a Boolean formula in CNF over a set of Boolean variables B. Let $\mathbb{D} = \bigcup_{i=1}^{n}(\bigcup_{j=1}^{k_i}\{\bar{l}_j^i\})$ be a domain universe; that is, a constant symbol with the stripe is introduced into $\mathbb{D}$ for each literal occurrence in F (notice that, each occurrence of a literal corresponds to a different constant symbol). The corresponding CSP (X, C, D) using literal encoding is built as follows: $X = \{\sigma_1, \sigma_2, \ldots, \sigma_N\}$; that is, a variable is introduced for each clause of F; it holds for $D: X \longrightarrow \mathcal{P}(\mathbb{D})$ that $D(\sigma_i) = \bigcup_{j=1}^{K_i}\{\bar{l}_j^i\}$; that is, the domain of an i-th clause contains constant symbols corresponding to all its literals. A constraint $c(\{\sigma_{i_1}, \sigma_{i_2}\}) = \langle(\sigma_{i_1}, \sigma_{i_2}), R^c\rangle$ is introduced over every pair of variables with $i_1, i_2 \in \{1,2,\ldots,N\} \wedge i_1 \neq i_2$ where a variable $x \in B$ such that either $x \in D(\sigma_{i_1}) \wedge \neg x \in D(\sigma_{i_2})$ or $\neg x \in D(\sigma_{i_1}) \wedge x \in D(\sigma_{i_2})$ exists. Such a constraint $c(\{\sigma_{i_1}, \sigma_{i_2}\})$ then forbids every tuple of values $(\bar{l}_{j_1}^{i_1}, \bar{l}_{j_2}^{i_2})$ such that there exists $x \in B$ for that either $\vec{l}_{j_1}^{i_1} = x \wedge \vec{l}_{j_2}^{i_2} = \neg x$ or $\vec{l}_{j_1}^{i_1} = \neg x \wedge \vec{l}_{j_2}^{i_2} = x$ (that is, the tuple $(\vec{l}_{j_1}^{i_1}, \vec{l}_{j_2}^{i_2})$ is removed from R^c which has been initially set to $D(\sigma_{i_1}) \times D(\sigma_{i_2})$). A solution of the resulting CSP (X, C, D) corresponds to the valuation of Boolean variables of B that satisfies F and vice versa [21].

Having the CSP model of SAT it is possible to check PC for the corresponding CSP model and proclaim the original SAT path-consistent or path-inconsistent accordingly. If elements of variable domains are interpreted as vertices and allowed tuples of values as directed edges connecting them, then PC with respect to a given path can be interpreted as existence of paths in the resulting directed graph. More precisely, let $P = (\sigma_{i_0}, \sigma_{i_1}, \dots, \sigma_{i_K})$ with $i_j \in \{1,2,\dots,N\}$ for $j = 0,1,\dots,K$ be a sequence of variables in the literal encoding CSP model (X, C, D). A directed graph $G_{\vec{PC}}(P) = (V, E)$, in which PC can be interpreted as the existence of paths, is defined as follows: $V = \bigcup_{j=0}^{K} D(\sigma_{i_j})$ and if $(\vec{l}_{j_1}^{i_j}, \vec{l}_{j_2}^{i_{(j+1) \bmod K}}) \in R^{c(\sigma_{i_j}, \sigma_{i_{(j+1) \bmod K}})}$ then a directed edge $(\vec{l}_{j_1}^{i_j}, \vec{l}_{j_2}^{i_{(j+1) \bmod K}})$ is included into E. A pair of values $\vec{l}_{j_1}^{i_0} \in D(\sigma_{i_0})$ and $\vec{l}_{j_2}^{i_K} \in D(\sigma_{i_K})$ is path-consistent with respect to the path P if there is an edge $(\vec{l}_{j_1}^{i_K}, \vec{l}_{j_1}^{i_0})$ in $G_{\vec{PC}}(P)$ and there exists a path from the vertex $\vec{l}_{j_1}^{i_0}$ to the vertex $\vec{l}_{j_2}^{i_K}$ in $G_{\vec{PC}}(P)$. The graph $G_{\vec{PC}}(P)$ will be called a *graph interpretation* of PC – see Fig. 2 for illustration.

Notice that PC is *incomplete* in the sense that a pair of values may be path-consistent even if there is no solution of the problem that contains this pair of values (see Fig. 2 again). Analogically, the problem may be path-consistent (that is, path-consistent with respect to all the paths) even if it has no solution actually. The partial reason for this weakness of PC is that many constraints are ignored when a pair of values is checked. This is especially apparent if a longer path of variables is considered. Only constraints over pairs of variables neighboring in the path are considered while many constraints such as that for example over the first and the third variable in the path are ignored. This property is disadvantageous especially in SAT where stronger reasoning is typically more beneficial.

For further augmentation of PC, it is also convenient to prepare a so called *auxiliary constraint graph* for the model with respect to the path P that reflects all the constraints over the variables of the path P. It is an undirected graph $G_{CSP}(P) = (V, E)$ and it is defined as follows: $V = \bigcup_{j=0}^{K} D(\sigma_{i_j})$; an edge $\{\vec{l}_{j_1}^{i_j}, \vec{l}_{j_2}^{i_k}\}$ is added to E if $(\vec{l}_{j_1}^{i_j}, \vec{l}_{j_2}^{i_k}) \notin R^{c(\sigma_{i_j}, \sigma_{i_k})}$; and all the edges $\{\vec{l}_{j_1}^{i_j}, \vec{l}_{j_2}^{i_j}\}$ for all $j = 1,2,\dots,N$ and $j_1, j_2 = 1,2,\dots,K_j \wedge j_1 \neq j_2$. Observe that the auxiliary constraint graph subsumes complement of the graph interpretation with respect to the same path.

4 Making Path-Consistency Stronger

A modification of PC has been proposed to overcome mentioned limitations of the standard version. To increase inference strength of PC additional requirements on the path in the graph model are imposed. These additional requirements reflect constraints over non-neighboring variables in the path of variables. As the auxiliary constraint graph represents an explicit representation of constraints, it is exploited for determining additional requirements.

An approach adopted in this work restricts the size of the intersection of the constructed path with certain subsets of vertices in the graph interpretation of PC. More precisely, let $G_{\vec{PC}}(P) = (V, E)$ be a graph interpretation of PC in a CSP model of SAT (X, C, D). The set of vertices V is partitioned into disjoint sequences $L_1, L_2, \ldots, L_M$ called *layers* (that is, $\bigcup_{i=1}^{M} \hat{L}_i = V$ and $\hat{L}_i \cap \hat{L}_j\ \forall i, j \in \{1,2,\ldots,M\} \wedge i \neq j$; where denotes the union of the sequence $\hat{A}$, that is $\hat{A} = \bigcup_{i=1}^{n}\{a_i\}$ for $A = [a_1, a_2, \ldots, a_n]$). The maximum size of the intersection of the path being checked to exist with individual layers is determined using the set of constraints C (notice that all the constraints over P are considered – not only constraints over neighboring variables in P). This proposal will be called an *initial augmentation* of PC in the rest of the text.

The concept of the initial augmentation of PC comes from [19]. The process of decomposition of the set of vertices into layers is done over the corresponding auxiliary constraint graph $G_{CSP}(P)$. Vertices of $G_{CSP}(P)$ are decomposed into vertex disjoint complete subgraphs. The knowledge of such decomposition can be then used to partition vertices into layers that directly correspond to found complete subgraphs. However, determining a complete subgraph is a difficult task itself. Hence a greedy approach has been used to obtain an acceptable solution.

Since it is possible to assign to a variable at most one value from values corresponding to vertices of the complete subgraph in $G_{CSP}(P)$, the maximum size of the intersection of the path with a layer is thus at most 1. Notice, that at most one value from vertices corresponding to the domain of a variable can be selected (this is due to the presence of the complete subgraph over the set of vertices corresponding to the domain of a variable in $G_{CSP}(P)$). Notice further, that if a value corresponding to a vertex in a complete subgraph is selected than all the values corresponding to other vertices of the complete subgraph are ruled out since they are in conflict with the selected value with respect to constraints.

A quite negative result has been obtained in [19]. It has been shown that finding a path, which conforms to the calculated maximum size of the intersection with individual layers, corresponds to finding a Hamiltonian path [5]. This is known to be an *NP*-hard problem. Hence, it is not tractable to find a path that satisfies defined requirements exactly. Moreover, initial experiments showed that it is almost impossible to make any reasonable relaxation of proposed requirements. Every relaxation of requirements on the path being constructed proposed by the author leads to weakening the modified PC down to the level of the standard version of PC (specifically, several adaptations of the algorithm for finding single source shortest paths [7] have been evaluated by the author).

These initial findings founded an effort to further augment requirements on the constructed path in order to allow developing stronger and more efficient relaxations. The result of this effort is a concept of a so called *modified version of PC*.

4.1 A Modified Version of Path-Consistency

Again, partitioning of vertices of $G_{\vec{PC}}(P)$ into layers is supposed. In addition, the sequencing of variables in the path P is exploited for defining the maximum size of the intersection of the constructed path with layers. Particularly, the path being constructed is required to conform to the calculated maximum size of the intersection with vertices of the layer preceding a given vertex of the path with respect to the sequencing of variables in P. The maximum size of the intersection is again imposed by the set of constraints C. More precisely, let $L_1, L_2, \dots, L_M$ be layers of $G_{\vec{PC}}(P)$; let a function $\chi: V \longrightarrow \mathbb{N}$ defines requirements on the maximum size of intersections imposed by constraints as follows: $\chi(v_j^l)$ is the maximum size of the intersection of the constructed path with a set of vertices $\{v_0^l, v_1^l, \dots v_i^l\}$ where $L_l = [v_0^l, v_1^l, \dots, v_{K^l}^l]$ with $l \in \{1,2,\dots,M\}$ and $j \in \{0,1,\dots,K^l\}$. Let a consistency defined by this new requirement on the constructed path be called a *modified PC*. Observe that this new concept is a generalization of the initial augmentation described above (see Fig 3 for illustration). It is intractable to construct a path conforming to the maximum sizes of intersections determined by χ as in the case of the initial augmentation. Nevertheless, it is possible to make a tractable relaxation of these requirements which does not collapse down to the level of the standard PC.

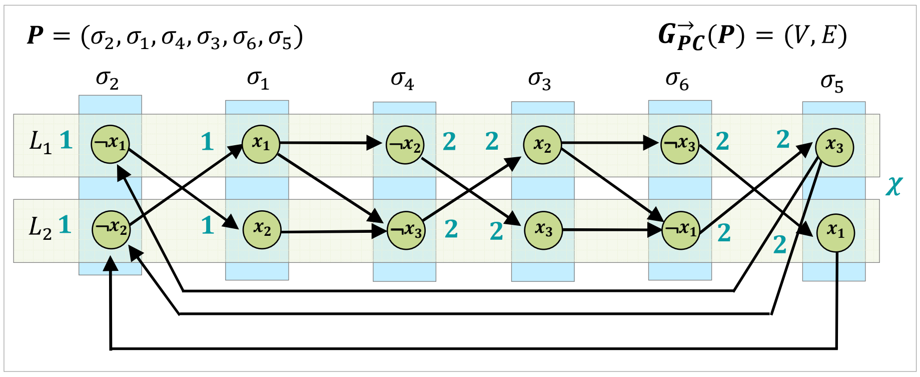

Fig 3. *An illustration of modified path-consistency in the CSP model of a SAT problem.* The maximum size of the intersection of the constructed path with vertices preceding the given vertex (including) in its layer is calculated using constraints for each vertex - these maximum sizes are denoted as the function χ. For example, having $\chi(\bar{l}_1^3) = 2$ then the constructed path can intersect the subset of vertices $\{\bar{l}_1^2, \bar{l}_1^1, \bar{l}_1^4, \bar{l}_1^3\}$ (first occurrences of literals in first four variables of the path P) of the layer L_1 in at most two vertices. Observe, that these requirements on the path being constructed rules out its existence for connecting a pair of vertices $\bar{l}_1^2$ from the left most variable (occurrence of literal $\neg x_1$) and $\bar{l}_1^5$ from the right most variable (occurrence of literal x_3). Compare it with the standard PC in Fig. 2 where the corresponding path connecting the same pair of vertices exists.

Let us now briefly describe such a tractable relaxation. Suppose that χ is already known (the process of calculation of χ will be described in the following section). Let $d_0 \in D(\sigma_{i_0})$ and $d_K \in D(\sigma_{i_K})$ be a pair of values for that a consistency is being checked. Two assignments will be maintained: $\Sigma: V \longrightarrow \mathbb{N}_0$ and $\psi: V \longrightarrow {\mathbb{N}_0}^{M\times(K+1)}$ where ${\mathbb{N}_0}^{M\times(K+1)}$ denotes matrices of the size $M \times (K+1)$ over $\mathbb{N}_0$. The assignment Σ will express the total number of distinct paths in $G_{\vec{PC}}(P) = (V, E)$ starting in d_0 and ending in a given vertex. Observe, that it is easy to calculate $\Sigma(v)$. It is determined recursively by the expression: $\Sigma(v) = \sum_{u \in V, (u,v) \in E} \Sigma(u)$, while $\Sigma(d_0) = 1$. The assignment ψ expresses statistical information about paths in $G_{\vec{PC}}(P)$ starting in d_0 and ending in a given vertex regarding the size of the intersection with layers. More precisely, an element of $\psi(v)$ at i-th row and j-th column (that is, $\psi(v)_{i,j}$) with $v \in V$, $i \in \{1,2,\dots,M\}$, and $j \in \{0,1,\dots,K\}$ represents the number of distinct paths starting in

d_0 and ending in v intersecting with the layer L_i in exactly j vertices that conform to relaxed requirements (that is, the size of the intersection of these paths with L_i is j). If the mentioned conformation to relaxed requirements is omitted, the information maintained in ψ is not difficult to be calculated recursively for every vertex of $G_{\vec{PC}}(P)$. However, as it is algorithmically more complex calculation, it is deferred to the section devoted to algorithms.

Requirements on the size of the intersection of the constructed path with layers represented by χ are relaxed in the following way. If it is detected that all the paths staring in d_0 and ending in v intersects the layer containing v in more vertices than it is allowed by χ, then it is possible to conclude that there is no path connecting d_0 and v that conforms to calculated maximum sizes of intersections with layers. Hence, v is unreachable from d_0 under given circumstances. The described relaxation can be expressed using defined assignments Σ and ψ. Let L_{l^v} be a layer containing v (that is, $v \in \hat{L}_{l^v}$). If there is some $j > \chi(v)$ such that $\psi(v)_{l^v,j} = \Sigma(v)$, then there is no path connecting d_0 and v conforming to the maximum sizes of intersection with layers. Observe, that although there is no $j > \chi(v)$ such that $\psi(v)_{l^v,j} = \Sigma(v)$, the required path still need not to exist. This is the principle which is called the relaxation in the context of this paper.

If it is detected that there is no path connecting d_0 and d_K that conforms to relaxed requirements on the maximum sizes of intersections with layers, the pair of values d_0 and d_K is said to be *inconsistent* with respect to the *modified PC*.

The detailed pseudo-code of greedy algorithms for determining layer decomposition and calculating maximum sizes of intersection is given in [20]. The pseudo-code of consistency enforcing algorithm also given there.

5 Augmenting Modified Path-Consistency to (2,*k*)-Consistency

Modified PC is trying to exploit more information from the constraints with scope containing a variable from the selected sequence with respect to which the consistency is checked. A question arises how strong the consistency would be if all the information from involved constraints is used; that is, if all the tuples of values forbidden by constraints are taken into account when a path connecting a pair of values is constructed. Such a consistency check corresponds to the concept of $(2, k)$-consistency [8] which is formalized in the following definition.

Definition 4 ($(2, k)$*-consistency*). Let (X, C, D) be a binary CSP and let $K = \{x_0, x_1, \dots, x_k, x_{k+1}\}$ with $x_i \in X$ for $i = 0,1, \dots, k+1$ be a set of distinct variables. A pair of values $d_0 \in D(x_0)$ and $d_{k+1} \in D(x_{k+1})$ is said to be consistent with respect to $(2, k)$-consistency and the set of variables K if there exists a valuation $v\colon \{x_0, x_1, \dots, x_k, x_{k+1}\} \longrightarrow \mathbb{D}$ with $v(x_0) = d_0 \wedge v(x_{k+1}) = d_k$ such that all the constraints $c(\{y, z\})$ such that $y, z \in K$ are satisfied by v. The CSP (X, C, D) is said to be $(2, k)$-consistent if it is $(2, k)$-consistent with respect to all the sets of size $k + 2$ and all the pairs of values. □

Intuitively, $(2, k)$-consistency checks for a pair of values from the domains of distinct variables whether there exists a k-tuple of values from other variables such that all these values form a consistent assignment; that is, a supporting k-tuple of consistent values is searched for a given pair of values (see Fig 4). The relation to (modified) PC is that enforcing $(2, k)$-consistency subsumes enforcing (modified) PC along paths of length $k + 2$. Notice that $(2, k)$-consistency represents the limit of the possible strength of any variant of modified PC. This fact has a consequence that if $(2, k)$-consistency eventually turn out to be unsuccessful when applied on SAT it is then useless to deal with the modified PC and make any further improvements of it. Fortunately, a part of the experimental evaluation showed that this is not the case.

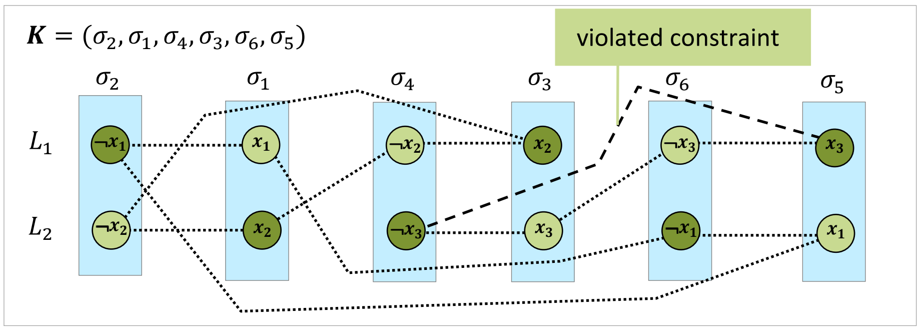

Fig 4. *An illustration of* $(2, k)$*-consistency for* $k = 4$ *in the CSP model of a SAT problem.* Forbidden pairs of values are depicted using dotted edges. The $(2, k)$-consistency can detect that a pair of literals $\neg x_1$ and x_3 is inconsistent since there is no set supporting consistent 4-tuple of values in σ_1, σ_2, σ_3, and σ_6. Notice that the path justifying standard PC of $\neg x_1$ and x_3 in Fig. 2 is forbidden here.

The application of higher order consistencies in SAT has been already studied. It has been shown in [16] that k-consistency can be simulated by clause learning mechanism within conflict-directed SAT solvers on a so called *direct encoding* of the given CSP instance. It is not known whether this is also true for $(2, k)$-consistency. The positive answer would also limit the effect of enforcing modified PC on situations where simulating the consistency by clause learning mechanism is inefficient. This is partially the reason why the modified PC is targeted on instances where the standard conflict-directed SAT solvers are inefficient.

6 Preliminary Experimental Evaluation

We have performed a preliminary experimental evaluation of the above suggestions. The evaluation was targeted on several aspects of the proposed concepts.

First, we investigated the quality of layer decomposition on standard SAT benchmarks. This is important for assessing the potential of modified PC. Next, we tried to identify a class of SAT instances where the described variant of the modified PC is beneficial when used as a preprocessing tool. The result of this evaluation is quite negative; it has shown that the current variant of modified PC can positively affect the performance of a SAT solver only on classical hard instances from [1]. These instances can be however solved by other techniques as well [1, 18].

The last aspect we have addressed in the evaluation is a question whether there exists a class of SAT instances where the modified PC after some improvements may be beneficial. The identification of such a class was done by applying $(2, k)$-consistency as a preprocessing tool. If $(2, k)$-consistency is successful on some SAT instance then there is a chance that an improvement of the modified PC may be also successful there. This experiment has shown that such a promising class of instances is represented by encodings of so called *difficult integer factorization* [2].

6.1 Experimental Implementation and Setup

There are several important issues regarding the implementation of the described method and the experimental setup. Consistencies which were tested as a preprocessing tool were first applied on a given instance for which they can infer new binary clauses. An augmented instance with additional binary clauses was then submitted to a SAT solver which eventually gave an answer to the instance. Several characteristics such as the number of inferred clauses, the number of decisions, and runtime were measured along the process. For our experiments, the Minisat2 SAT solver [10] was used.

To improve inference strength of preprocessing the *singleton unit propagation* has been performed first [9] (that is, each literal is assigned the value $TRUE$ and unit propagation is performed; assignments to literals enforced by the propagation together with assignment to the original literal form additional constraints). It can discover some binary clauses that can be used in the literal encoding of the SAT instance as implied constraints. These implied constraints subsequently reduce the size of the intersection with layers in modified PC.

As it is computationally too expensive to enforce modified PC as well as $(2, k)$-consistency with respect to all paths and subsets of variables respectively only some promising paths and subsets of variables were used. The length of the sequence of variables varied from 4 to 8. For each length of the sequence of variables and for each clause of the instance, the most promising sequence of variables starting in the given clause was determined and consistency was calculated with respect to it. That is, the existence of a path or a consistent k-tuple was checked for all the pairs of values from the first and the last variable of the selected sequence.

When constructing the most promising sequence of variables/clauses, a clauses that is most constrained with respect to the currently last clause was added with the probability of 0.7 otherwise a randomly selected clause has been added.

Although the given consistency was computed in a partial manner using the above approach, it represents a suitable trade-off between the number of inferred clauses and the effort to compute them.

All the tested algorithms were implemented in C++. The tests were run on an dual AMD Opteron 1600 MHz, with 1GB RAM under Mandriva Linux 10.1, 32-bit edition, gcc 3.4.3 was used for compilation.

6.2 Tests of Modified Path-Consistency

The first part of the evaluation of modified PC is devoted to getting a visual insight of the consistency on standard satisfiability benchmarks from SATLib [12].

Table 1. *Maximum intersection sizes with the first layer of the layer decomposition*. The intersection sizes are calculated in the graph interpretation of several satisfiability instances from SATLib using the greedy algorithm described in [20].

SAT instance	Maximum **intersection** with $L_1 = [v_0^1, v_1^1, \dots, v_7^1]$							
	$\chi(v_0^1)$	$\chi(v_1^1)$	$\chi(v_2^1)$	$\chi(v_3^1)$	$\chi(v_4^1)$	$\chi(v_5^1)$	$\chi(v_6^1)$	$\chi(v_7^1)$
ais12.cnf	1	1	1	1	1	1	1	1
hanoi4.cnf	1	2	2	3	3	3	4	4
huge.cnf	1	1	2	2	2	2	3	3
jnh1.cnf	1	2	2	3	4	4	4	5
par16-1.cnf	1	1	1	2	2	2	2	2
par16-1-c.cnf	1	2	2	3	3	4	4	5
pret150_75.cnf	1	1	2	2	3	3	4	4
s3-3-3-8.cnf	1	1	2	3	3	4	4	5
ssa7552-160.cnf	1	1	2	3	4	4	5	6
sw100-5.cnf	1	1	2	2	2	2	3	3
Urq8_5.cnf	1	1	2	2	3	3	4	4
uuf250-0100.cnf	1	1	2	2	3	3	4	4

Results regarding maximum sizes of intersections with layers calculated by greedy algorithms from [20] are shown in Table 1. Parts of the auxiliary constraint graph restricted on the first layer of several instances are shown in Fig 5.

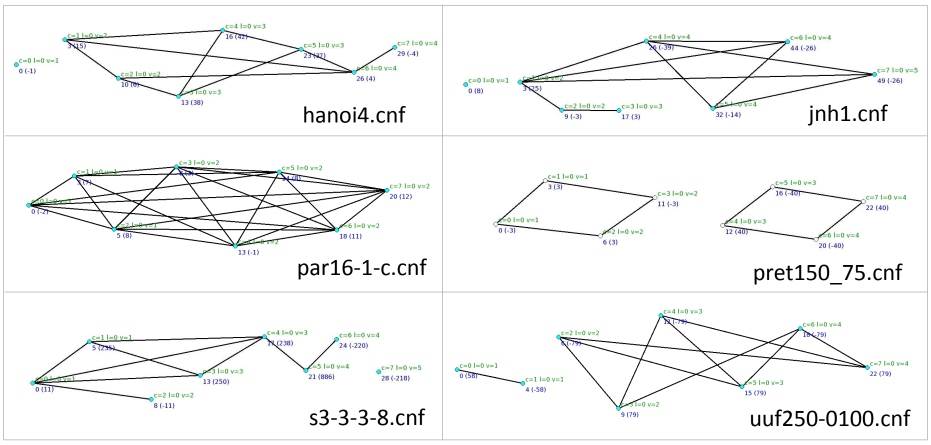

Fig 5. *An illustration of first layers of the auxiliary constraint graph of several satisfiability instances.* Several sparse and dense graphs are shown together with calculated maximum sizes of intersection of the path being constructed with the layer.

These results indicate that information captured by constraints over non-neighboring variables in the sequence of variables is relatively strongly reflected in the maximum sizes of intersections with layers. It is evident that the calculation of maximum sizes of intersections is more successful (that is, the resulting allowed size of the intersection is smaller) on instances with dense auxiliary constraint graphs. The extreme case is when a layer is represented by a complete graph with maximum size of the intersection equal to 1. Consequently, modified PC is most successful in such cases.

Table 2. *Comparison of the standard path-consistency (PC) and modified path-consistency (mPC) in terms of* ***inference strength***. The number of newly inferred clauses by both tested consistencies on difficult SAT instances [1] is reported. The comparison of preprocessing abilities is shown in terms of the number of decisions made by the Minisat2 SAT solver of instances augmented by inferred clauses (N/A indicated that the result was not obtained due to the timeout of 10.0 seconds).

SAT instance	**Instance characteristics**		**Inferred binary clauses**		**Minisat2 decisions**		
	Variables	Clauses	PC	mPC	Original	PC	mPC
hole6	42	133	0	42	1777	1777	1
hole7	56	204	0	56	10123	10123	1
hole8	72	297	0	72	40554	40554	1
hole9	90	415	0	90	202160	202160	1
chnl10_11	220	1122	0	220	N/A	N/A	1
chnl10_12	240	1344	0	240	N/A	N/A	1
chnl10_13	260	1586	0	260	N/A	N/A	1
chnl11_12	264	1476	0	264	N/A	N/A	1
chnl11_13	286	1742	0	286	N/A	N/A	1
chnl11_20	440	4220	0	440	N/A	N/A	1
fpga10_12_uns_rcr	240	1344	0	240	N/A	N/A	1
fpga10_13_uns_rcr	260	1586	0	260	N/A	N/A	1
fpga10_15_uns_rcr	300	2130	0	300	N/A	N/A	1
fpga10_20_uns_rcr	400	3840	0	400	N/A	N/A	1
fpga11_11_uns_rcr	264	1476	0	264	N/A	N/A	1
fpga11_12_uns_rcr	286	1742	0	286	N/A	N/A	1

Table 3. *Comparison of the standard path-consistency (PC) and modified path-consistency (mPC) in terms of* ***runtime***. Runtimes necessary for preprocessing, solving original and preprocessed instance, and total runtime (preprocessing + solving) are reported.

SAT instance	**Preprocessing runtime** (sec.)		**Minisat2 solving runtime** (sec.)			**Total solving runtime** (sec.)	
	Runtime PC (sec.)	Runtime mPC (sec.)	Original	PC	mPC	PC	mPC
hole6	0.01	0.04	0.00	0.00	0.00	0.01	0.04
hole7	0.02	0.14	0.10	0.10	0.00	0.12	0.14
hole8	0.04	0.32	0.48	0.48	0.00	0.52	**0.32**
hole9	0.07	0.64	3.61	3.61	0.00	3.68	**0.64**
chnl10_11	0.23	2.38	> 10.0	> 10.0	0.00	> 10.0	**2.38**
chnl10_12	0.25	2.6	> 10.0	> 10.0	0.00	> 10.0	**2.6**
chnl10_13	0.27	2.82	> 10.0	> 10.0	0.00	> 10.0	**2.82**
chnl11_12	0.36	4.18	> 10.0	> 10.0	0.00	> 10.0	**4.18**
chnl11_13	0.39	4.54	> 10.0	> 10.0	0.00	> 10.0	**4.54**
chnl11_20	0.63	7.05	> 10.0	> 10.0	0.00	> 10.0	**7.05**
fpga10_12_uns_rcr	0.25	2.61	> 10.0	> 10.0	0.00	> 10.0	**2.61**
fpga10_13_uns_rcr	0.28	2.82	> 10.0	> 10.0	0.00	> 10.0	**2.82**
fpga10_15_uns_rcr	0.32	3.27	> 10.0	> 10.0	0.00	> 10.0	**3.27**
fpga10_20_uns_rcr	0.45	4.37	> 10.0	> 10.0	0.00	> 10.0	**4.37**
fpga11_11_uns_rcr	0.36	4.18	> 10.0	> 10.0	0.00	> 10.0	**4.18**
fpga11_12_uns_rcr	0.39	4.54	> 10.0	> 10.0	0.00	> 10.0	**4.54**

Instances where this phenomenon can be observed are shown in Table 2 and Table 3. These results show how the modified PC can be successful as preprocessing tool when applied on well known difficult SAT instances which auxiliary constraint graphs can be decomposed into layers forming complete graphs. A significant improvement was achieved using modified PC in terms of runtime as well as in terms of the number of decisions made the SAT solver. The comparison with standard PC is shown for reference – expectably it is unable to infer any new clause.

Although additional experimental evaluation showed that modified PC can infer many new clauses and reduce the number of decisions of the SAT solver on other types of instances (for example that from Table 1) [20], current implementation has too long runtime to be successfully applied as a preprocessing tool these instances in a greater scale.

6.2 Tests of (2,*k*)-Consistency

Limits how modified PC can be further improved were explored using $(2,k)$-consistency. Regarding the inference strength $(2,k)$-consistency any improvement of modified PC cannot infer more than $(2,k)$-consistency. Hence, if there is no relevant set of instances where $(2,k)$-consistency significantly more clauses than PC, then modified PC has even less chance. Results shown in Table 4 and Table 5 indicate not only that this is not the case but even that $(2,k)$-consistency itself can successfully used as a preprocessing tool in spite of the fact that it is computationally expensive. The class of instances where the preprocessing was successful encodes difficult integer factorization problems [2].

Table 4. *Comparison of the standard path-consistency (PC) and (2,k)-consistency ((2,k)-c) in terms of* ***inference strength***. The number of newly inferred clauses on difficult integer factorization instances [2] is reported. The performance in terms of the number of decisions made by the Minisat2 solver on original and preprocessed instances is shown.

SAT instance	Instance characteristics		Inferred binary clauses		Minisat2 decisions		
	Variables	Clauses	PC	(2,k)-c	Original	PC	(2,k)-c
difp_19_0_arr_rcr	1201	6563	0	675	142710	142710	61317
difp_19_0_wal_rcr	1755	10446	103	281	73018	339662	25340
difp_19_1_arr_rcr	1201	6563	6	307	250692	87894	81144
difp_19_1_wal_rcr	1755	10446	363	561	129235	133055	77039
difp_19_2_wal_rcr	1755	10446	38	212	288500	207775	98374
difp_19_3_arr_rcr	1201	6563	128	342	114648	122379	100824
difp_19_3_wal_rcr	1755	10446	36	202	609247	968223	109741
difp_20_0_arr_rcr	1201	6563	91	754	8174	39097	12598
difp_20_0_wal_rcr	1755	10446	378	553	65601	752497	123562
difp_20_1_wal_rcr	1755	10446	10	131	362145	540378	147005
difp_20_2_arr_rcr	1201	6563	57	611	62119	438572	49700
difp_20_2_wal_rcr	1755	10446	866	2375	184778	177142	15415
difp_20_3_arr_rcr	1201	6563	0	73	142823	142823	89801
difp_20_3_wal_rcr	1755	10446	357	5798	26905	159962	45492

It is often the case that a preprocessing influences the SAT solver's performance in an unpredictable way. Hence it is another important point here that the improvement of the number of decisions and runtime seems to be relatively stable on preprocessed instances from the given class.

Table 5. *Runtime Comparison of the standard path-consistency (PC) and (2,k)-consistency ((2,k)-c) in terms of* ***runtime***. Runtimes for preprocessing, solving original and preprocessed instance, and total runtime (preprocessing + solving) are reported.

SAT instance	**Preprocessing time** (sec.)		**Minisat2 solving time** (sec.)			**Total solving time** (sec.)	
	Runtime PC	Runtime (2,k)-c	Original	PC	(2,k)-c	PC	(2,k)-c
difp_19_0_arr_rcr	3.15	3.19	27.78	27.78	9.63	30.93	**12.82**
difp_19_0_wal_rcr	3.74	3.58	10.97	68.94	2.68	72.68	**6.26**
difp_19_1_arr_rcr	3.00	3.06	54.17	15.99	14.21	**18.99**	**17.27**
difp_19_1_wal_rcr	3.56	3.48	24.19	24.86	14.21	28.42	**17.69**
difp_19_2_wal_rcr	3.58	3.43	65.13	41.53	18.85	**45.11**	**22.28**
difp_19_3_arr_rcr	3.00	3.57	20.59	22.80	16.86	25.80	**20.43**
difp_19_3_wal_rcr	3.79	3.48	164.05	286.71	19.59	290.50	**23.07**
difp_20_0_arr_rcr	3.04	3.43	0.73	5.69	1.11	8.73	12.16
difp_20_0_wal_rcr	3.64	3.62	10.48	208.25	23.45	211.89	27.07
difp_20_1_wal_rcr	3.99	3.69	83.49	134.27	28.22	137.96	**31.91**
difp_20_2_arr_rcr	3.01	3.36	9.57	108.28	7.40	111.29	10.76
difp_20_2_wal_rcr	23.3	25.6	38.49	37.47	1.62	60.77	**27.22**
difp_20_3_arr_rcr	17.33	19.6	27.73	27.73	16.78	45.06	36.38
difp_20_3_wal_rcr	21.94	23.8	3.54	31.27	7.33	53.21	31.13

7 Conclusion and Future Work

The recently proposed concept of consistency for Boolean satisfiability called modified PC has been revisited in this paper. The new type of consistency augments the standard PC by exploiting global properties of the input instance. Particularly, stronger requirements are imposed on the path being checked to exist.

The experimental evaluation where modified PC is used as a SAT preprocessing tool discovered that it is especially successful on hard satisfiability instances with structured constraint network [1]. Unfortunately, modified PC is not so successful on other instances; the preprocessing runtime typically nullify all the eventual benefit gained by the shorter runtime of the SAT solver.

Another experimental evaluation was targeted on assessing possible limits of improvement of the modified PC by running $(2,k)$-consistency on the selected set of SAT instances. This experiment showed that even relatively computationally costly $(2,k)$-consistency can serve as an efficient preprocessing tool.

There is still lot of work for the future. It is necessary to improve implementation of the modified PC and to evaluate it on larger set of benchmark instances. Nevertheless, the most desirable goal is to make improvement of modified PC so that it can be better alternative than both standard PC and $(2,k)$-consistency.

References

1. Aloul, F. A., Ramani, A., Markov, I. L., Sakallah, K. A.: Solving Difficult SAT Instances in the Presence of Symmetry. Proceedings of the 39th Design Automation Conference (DAC 2002), 731-736, USA, ACM Press, 2002, http://www.aloul.net/benchmarks.html, [March 2011].

2. Aloul, F. A.: SAT Benchmarks, Difficult Integer Factorization Problems. Research web page, http://www.aloul.net/benchmarks.html, [March 2011].
3. Chmeiss, A., Jégou, P.: Two New Constraint Propagation Algorithms Requiring Small Space Complexity. Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI 1996), pp. 286-289, IEEE Computer Society, 1996.
4. Chmeiss, A., Jégou, P.: Efficient Constraint Propagation with Good Space Complexity. Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP 1996), pp. 533-534, LNCS 1118, Springer, 1996.
5. Chvátal, V.: Tough Graphs and Hamiltonian Circuits. Discrete Mathematics 306, Volume 10-11, pp. 910-917, Elsevier, 2006.
6. Cook, S. A.: The Complexity of Theorem Proving Procedures. Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151-158, ACM Press, 1971.
7. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.: Introduction to Algorithms (Second edition), MIT Press and McGraw-Hill, 2001.
8. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, 2003.
9. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming, Volume 1 (3), 267-284, Elsevier Science Publishers, 1984.
10. Eén, N., Sörensson, N.: MiniSat — A SAT Solver with Conflict-Clause Minimization. Poster, 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005), 2005.
11. Han, C. C., Lee, C. H.: Comments on Mohr and Henderson's Path Consistency Algorithm. Artificial Intelligence, Volume 36(1), pp. 125-130, Elsevier, 1988.
12. Holger, H. H., Stützle, T.: SATLIB: An Online Resource for Research on SAT. Proceedings of Theory and Applications of Satisfiability Testing, 4th International Conference (SAT 2000), pp.283-292, IOS Press, 2000, http://www.satlib.org, [March 2011].
13. Jackson, P., Sheridan, D.. Clause Form Conversions for Boolean Circuits. Theory and Applications of Satisfiability Testing, 7th International Conference (SAT 2004), Revised Selected Papers, pp. 183–198, Lecture Notes in Computer Science 3542, Springer 2005.
14. Mohr, R., Henderson, T. C.: Arc and Path Consistency Revisited. Artificial Intelligence, Volume 28 (2), 225-233, Elsevier Science Publishers, 1986.
15. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences, Volume 7, pp. 95-132, Elsevier, 1974.
16. Petke, J., Jeavons, P.: Local Consistency and SAT-Solvers. Proceedings of Principles and Practice of Constraint Programming, 16th International Conference (CP 2010), pp. 398-413, LNCS 6308, Springer, 2010.
17. Rossi, F., Dhar, V., Petrie, C.: On the Equivalence of Constraint Satisfactions Problems. Proceedings of the 9th European Conference on Artificial Intelligence (ECAI 1990), pp. 550-556, 1990.
18. Surynek, P.: Solving Difficult SAT Instances Using Greedy Clique Decomposition. Proceedings of the 7th Symposium on Abstraction, Reformulation, and Approximation (SARA 2007), LNAI 4612, pp. 359-374, Springer, 2007.
19. Surynek, P.: Making Path Consistency Stronger for SAT. Proceedings of the Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2008), ISTC-CNR, 2008.
20. Surynek, P.: An Adaptation of Path Consistency for Boolean Satisfiability: a Theoretical View of the Concept. Proceedings of the Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2010 (CSCLP 2010), pp. 16-30, Fraunhofer FIRST, 2010.
21. Walsh, T.: SAT vs. CSP. Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, 441-456, LNCS 1894, Springer Verlag, 2000.

Analysis of Basic Heuristic Actions: Extensions and Clarifications

Richard J. Wallace

Cork Constraint Computation Centre and Department of Computer Science
University College Cork, Cork, Ireland
email: r.wallace@4c.ucc.ie

Abstract. In this paper we begin to build on an earlier proposal, that variable ordering heuristics for CSPs have two fundamental effects, called contention and simplification. These effects were deduced from the study of statistical variation in performance across a set of problems when different heuristics were employed. In the present work, we attempt to describe these two forms of action analytically, building on earlier probabilistic analyses of [HE80] and, especially, of [SG98]. We then attempt, still with only limited success, to account for differences among individual problems in their amenability to the two forms of heuristic action. We then discuss the important idea of "heuristic synergy", which occurs when the two basic heuristic actions are balanced to an appropriate degree. Finally, we briefly discuss the relations of this analysis to the Policy Framework for describing heuristic performance, and whether this analysis can be extended successfully to problems with various kinds of structural features.

1 Introduction

The purpose of the present paper is to consider some recent results on the nature of variable ordering heuristics for CSPs and work out some of their implications. The ultimate aim is to produce a theory of heuristics, at least in the context of constraint satisfaction problems (CSPs). Such a theory should give a coherent explanation of what these heuristics are actually doing and should explain why certain heuristics are so effective in reducing search effort.

One of the difficulties in making sense of this area is that variable ordering heuristics come in so many shapes and sizes. There are those, like maximum static degree, that depend on the basic topology of the problem. Others, like minimum domain size, are highly dependent on the choices made before the present point of selection. Others, like minimum domain-size/degree, are compounded out of more than one problem feature. Still others, such as the weighted degree heuristics, are critically dependent on information gathered during search.

It is possible to make some tentative classifications based on the features used by the heuristic, e.g. degree-based heuristics vs. domain-size-based heuristics. A more general example is static versus dynamic heuristics (respective examples are degree in the original constraint graph and current domain size). However, although these seem to capture some important differences, in many cases it is not at all clear how these categories should be applied. For example, is minimum domain/static-degree a domain or degree heuristic? And how does it fit into the static/dynamic classification?

A more subtle problem is that these simple classifications are based on the kinds of information used by the heuristic, not on what it does. In fact, until recently, there was perhaps only one paper that concerned itself with characterising heuristics in terms of what they do; this was the paper by Hooker and Vernay [HV95] on branching heuristics for SAT problems. (Branching heuristics is another term for variable ordering heuristics.) They considered two possibilities, satisfaction (increasing the likelihood that the [sub]problem will be satisfied) and simplification (essentially search space reduction), and adduced evidence and arguments that the latter was the important factor.

Recently, further progress has been made by applying techniques of statistical classification to this problem. Patterns of performance across a set of (homogeneous) random binary CSPs having the same basic parameter values (i.e. the same number of variables, initial domain size, constraint graph density and constraint tightness) and for the same algorithm, but with twelve different heuristics, were subjected to factor analysis. By combining this technique with various experimental manipulations, it was possible to show that for these problems variable ordering heuristics fall into two general classes [Wal05,Wal08]. Specifically,

- A factor analysis model with only two factors was able to account for up to over 90 percent of the variance (depending on the quality of controls, such as using repeated runs for each problem with random value selection).
- All heuristics tested were found to load more heavily on (i.e. show higher correlations with) one or the other of the two factors. Thus, heuristics could be classified according to the factor they ‘favoured’.

With respect to the second point, it is critical to note that there were usually definite positive loadings on both factors (typically, .45 versus .85 for a given heuristic).

Now, since all that factor analysis does is extract patterns of variation, further analysis (both conceptual and experimental) is necessary to determine the cause of this variation and whether it is of *scientific* significance. In the present case, there is compelling evidence that these two factors reflect two distinct *heuristic actions*. (In this case, the distinctness is also indicated by the fact that factor analysis in its usual form extracts *uncorrelated* factors.)

The most important evidence for this comes from experiments in which heuristics were formed based on additive combinations of rankings derived from two or more of the basic heuristics used in the original experiments [Wal06a]. It was shown that combinations of heuristics that loaded most heavily on *different* factors served to enhance search, so that performance was better than the best heuristic used singly. This effect was called “heuristic synergy”. If the heuristics favoured the same factor, the results were in between those for the heuristics used singly. Moreover, synergistic effects were as great for appropriate combinations of two heuristics as for combinations of three or more, which supports the hypothesis that there are only two basic kinds of heuristic action for these problems (and possibly only two *fundamental* heuristic actions in general).

Further evidence comes from the fact that the two 'kinds' of heuristic have different performance signatures [Wal08]. That is, measures such as average depth of failure or average branching factor show differences that are statistically distinguishable.

Finally, it is possible to associate one of these two heuristic actions with the simplification factor of [HV95]; hence, it is referred to as the "simplification factor". The other factor is referred to as the "contention factor" [Wal08]. Another way of describing these actions is to say that the latter focuses on immediate failure (i.e. failure of the current variable), while the former focuses on future failure [Wal05,vB06].

Note that the proposal is not that there are two basic classes of heuristics, although an important question is why heuristics tend to fall into two classes with respect to the basic heuristic actions. In fact, a heuristic based on more than one problem feature may be more highly associated with one or the other action depending on the kind of problem. Thus, two heuristics in the FF series of [SG98], FF2 and FF3, are contention heuristics when used on random CSPs and simplification heuristics when used on random k-colouring problems [Wal08].

Although this work (in my opinion) constituted a major advance in our understanding of variable ordering heuristics, many questions were left open. These include the following.

- What is the basis for distinct heuristic actions?
- What is the basis for the characteristic signature of each of the two types of heuristic action?
- Can measures be found that can distinguish random problems with respect to their relative amenability to the two forms of heuristic action?
- Why is balancing heuristic actions so effective?
- Why do adaptive heuristics facilitate this balancing?

In the present paper, we begin to answer some of these questions. The next section discusses the first two from the list above, the next section discusses the third, and the section after that discusses the last two questions. The section after that contains a brief discussion of the relation of the heuristic actions to the policy framework of [BPW03,BPW05,Wal06b]. The penultimate section discusses whether there is life for this account beyond random problems. Some concluding remarks are made in the final section.

2 Basis for Heuristic Actions

The effect of choosing the most contentious variable can be understood most easily by considering the simplest strategy to this effect, which is to choose the variable with the smallest current domain size. For simplicity, suppose that for each value in a domain the probability of failure has the same value p equal to 1/2. In this case, for a domain of size 2 the probability of all values failing is 1/2 * 1/2 or 1/4. For a domain of size 3, the probability is $(1/2)^3$ or 1/8. So, clearly, the minimum domain size heuristic will, other things being equal, be more likely to result in failure.

Now, if one were able to gauge likelihood of failure more precisely, it might be possible to choose variables with more than the minimum domain size that still yield a greater likelihood of failure. The more complex "fail-first" (FF) heuristics do just that. However, the basic strategy is still the same: choose the variable where the likelihood of immediate failure is greatest.

Simplification offers a second powerful strategy for improving search. Again the strategy can be understood by considering the simplest strategy of this type, which is to choose the variable with the highest forward degree. When this is done, an assignment to the selected variable, x, has direct effects on the maximal number of future variables; this also yields a maximal number of second-order effects, etc. The basic idea is to constrain the future subgraph in as many places as possible, which increases the probability of finding a 'weak link'. When this strategy is used at successive levels of search, the likelihood that some future variable is affected by a sufficient number of past variables increases, thus increasing the probability that something will fail.

At first blush the preceeding explanation does not seem to account for the similar patterns of performance across problems of degree heuristics when simple backtracking is used, i.e. when there is no propagation [Wal08]. However, by setting up multiple potential fault-points, one is essentially employing the same strategy, although in this case failure will occur only after the weak link is reached during search, i.e. when it is the current variable. As noted in [Wal08], this issue is of considerable theoretical importance because if the present account is true, it shows that simplification is not just a matter of domain reduction, but is more like what Gomes has called "unlocking the combinatorics of the problem".

These arguments can be stated more formally; in doing so, several issues are clarified that are obscured by the loose intuitive account just given. Since the following argument leaves out some of the features of the situation, it is still incomplete. It is also limited at present to binary constraints.

As noted above, a heuristic like minimum domain size increases the likelihood of failure at the present node in the search tree. This argument was elaborated by [SG98]. Their analysis is more apposite in the present context than the earlier fail-first analysis of [HE80], because it focuses on the current variable to be selected for assignment, while the latter developed their arguments in terms of expected depth of failure. They also avoid a simplifying assumption that limits the applicability of the earlier analysis, namely that the probability that an assignment will fail is the same for all values of all unassigned variables. (Incidentally, this is not to say that the analysis of [SG98] supersedes that of [HE80] in all respects; in fact, in certain aspects it depends on the latter for its association with the Fail First Principle. In fact, this analysis is more pertinent to contention in the present sense than to fail-firstness, which is a broader concept (cf. Section 5).)

The argument of [SG98] begins with the observation that, given certain independence assumptions, the probability that a selected variable v_i fails is the product of the probability of each assignment to v_i failing. In addition, an assignment x to v_i fails if there is some constraint with variable v_j such that there is no value in the domain of v_j consistent with x. This can be expressed by considering the alternative case, where at least one label of v_j is consistent, viz,

$$(1 - p_{ij}^{m_j})$$

where p_{ij} is the tightness of the constraint between v_i and v_j and m_j is the size of the domain of v_j (again assuming some independence conditions). Then the probability that x will fail is 1 minus the product of all such probabilities across all constraints between v_i and future variables,

$$1 \; - \prod_{v_j \in F, j \neq i} (1 - p_{ij}^{m_j})$$

And since, as already noted, probabilities are independent across values of v_i, the following formula gives the probability of v_i failing:

$$(1 \; - \prod_{v_j \in F, j \neq i} (1 - p_{ij}^{m_j}))^{m_i} \tag{1}$$

The pertinence of this analysis in the present context follows when we consider that the heuristics based on this analysis (the FF series) all load most heavily on the same factor, the one associated here with a "contention" strategy or action. This means that "contention" as used here can be associated with failure across the constraints adjacent to the current variable. At the same time, one should not lose sight of the fact that the conditions under which equation (1) is maximised by selecting a given variable depend heavily on the existing instantiations. (In this respect, therefore, the argument is still incomplete.) It is because of this that the contention strategy can be said to be a strategy of immediate failure rather than future failure; in other words, the strategy always involves focusing on the current variable.

In describing simplification effects, we will focus on two subsets of variables: the currently *assigned variables* and the (current) *future-adjacent variables*, which are those variables in the set of unassigned variables that are adjacent to at least one assigned variable. The reason for emphasizing the second set rather than the full set of future variables is that the former is much more likely to be 'the seat of the action' due to its adjacency to variables whose domains have all been reduced to one value.

In order to describe simplification effects, we first note that simplification heuristics tend to increase the size of the future-adjacent subgraph as much as possible. In other words, a larger number of variables are adjacent to variables whose current domain has been reduced to one value. This has several effects. In the first place, the probability of failure across a constraint between a variable in the instantiated set and one in the future set is increased. For example, suppose that the expected probability of success across a constraint is p, so the probability of failure is $1 - p$. If there is an additional constraint, where the probability of success is q, then the probability of failure is $1 - pq$. For a third constraint with probability of r, the probability of failure is $1 - pqr$, etc.

A second effect concerns the probability of failure within the set of unassigned variables, in particular the future adjacent set. To derive an expression for failure within the future adjacent set, we consider that the size of this set is k and there are c constraints between pairs of variables in this set. For simplicity we will assume a constant domain size d and tightness p_2. Then the expected number of consistent partial assignments to members of this set is $d^k(1 - p_2)^c$. (Note that we can retain this expression whilst considering the remaining varibles in the future set, viz,

$$d^k(1-p_2)^c \; * \; d^{k'}(1-p_2)^{c'} \; * \; (1-p_2)^{c''}$$

where k' and c' are the number of variables and constraints, respectively, in the last-mentioned set and c" is the number of constraint between future-adjacent and other future variables.)

Now, for a given k, as the number of adjacent assignments increases, the value for d^k decreases. As a result, the value for $d^k(1 - p_2)^c$ also decreases. Moreover, as the size of the future-adjacent set increases, k increases linearly while the increase in c is generally $O(n^2)$. In the latter case, as k increases k and c will diverge without limit. And since $1 - p_2$ and d are both constants and $0 < 1 - p_2 < 1$, $d^k(1 - p_2)^c$ will eventually tend to 0. This is most striking when $c = k(k - 1)/2$ as shown in the following table, but clearly it will hold for any density that can be described by $\alpha(k(k - 1)/2$, where α is a constant ≤ 1.

k	c	$d^k(1 - p_2)^c$
2	1	60
3	3	216
4	6	467
5	10	605
6	15	470
7	21	219
8	28	61
9	36	10
10	45	1

Note that the first effect described above for simplification is similar to contention in that it pertains to the constraints adjacent to the current variable. Despite this, since the selection criteria are different, it will result in a different pattern of failure. However, neither this effect nor the second is completely independent of an enhancement-of-failure strategy like min domain size, as long as there are constraints with both past and future variables. This begins to account for the fact that heuristics always show clear-cut positive loadings on both factors.

3 Differences among Problems Wrt Effects of Different Heuristic Actions

An interesting and potentially important discovery that emerged from the factor analysis work is that different problems are differentially amenable to the different heuristic actions. That is, within an ostensibly homogeneous set of problems, i.e. a set having the same basic parameters, it is found that with some problems contention heuristics perform better than simplification heuristics, while for other problems the difference is reversed. Presumably, this difference is related to the conditions described in the last section. (Note that while an actual performance reversal is not strictly implied by the factor analysis, it is observed in practice.)

Here it should be mentioned that some experimental work was done earlier to test the hypothesis that the effectiveness of simplification depended on the interaction of variables mutually affected by constraint propagation. This idea was tested using problems whose basic constraint graph was a torus (a grid folded back on itself, so each variable has four neighbors). Additional constraints were added to the torus structure so that all were adjacent to 2-3 (distinguished) variables. Under different conditions, the maximum distance in the graph from the distinguished variable to the adjacent variable was varied, as well as the number of extra constraints added (and the distance between the distinguished variable themselves). In this way the nearness in the graph between variables adjacent to different distinguished variables, and hence the presumed degree of interaction, could be varied. In this situation, the relative effectiveness of contention and simplification heuristics was altered in a way that could be predicted from these nearness relations [Wal08].

In the present work, the idea was to examine ordinary (homogeneous) random problems, to see if these could be distinguished in a way that could be related to performance differences. To this end, tests were run in which measures were taken that might be expected to discriminate between the two kinds of problem, given the arguments developed in the previous section. These preliminary tests made use of the same 100 50-variable random problems that were used in some of the earlier factor analysis studies. In terms of the basic $< n, d, p_1, p_2 >$ scheme (n= number of variables, d= domain size [in this case constant],p_1=graph density, and p_2= constraint tightness [also constant in this case]), the problems were <50,10,0.184,0.369>. Tests were run by using the maximum forward degree heuristic, which is a simplification heuristic. For each problem, there was one run through the set of variables.

The basic idea was to examine correlations between graph measures and the difference in performance (here, search nodes) when either a simplification or a contention heuristic was used. For this purpose, two heuristics were chosen (max forward degree (fd) and FF2) that were comparable in their mean performance across this set of problems. Here, mean performance was based on means for each problem over 100 runs with random value selections; these data were used to ensure that differences between fd and FF2 were not due to serendipitous discovery of a viable assignment early in the value ordering for the domain of the variable just chosen. The (grand) means for search nodes per problem across all problems were 3414 and 2732 for FF2 and fd, respectively, when MAC-3 was used as the basic algorithm.

In order to obtain a score showing the difference in performance between the two heuristics, the following formula was used (n_x designates search nodes for heuristic x):

$$(n_{FF2} - n_{fd})/\max(n_{FF2}, n_{fd})$$

This gives the degree of improvement as a proportion of the larger value for search effort. Scores range between 1.0 and -1.0, where positive scores indicate that fd is superior (fewer nodes). For the statistical analysis, scores on this bipolar scale were transformed into ranks, where a lower rank indicated a proportionally greater difference in favour of fd.

(An incidental note: when a similar analysis was done on the data from single runs per problem with lexical value ordering, it was found that the correlation between the two sets of ranks was only 0.34, indicating that happening on viable values does serve to obscure basic performance differences for these problems.)

Three kinds of tests were run; in each case the results were correlated with the performance measure just described. The idea in this preliminary work was to see if a relevant measure could be found that showed some indication of being correlated with performance.

In the first set of experiments, the number of adjacent assigned variables was counted after each successive variable selection, i.e. at each depth of search. For clarity, an example of the output is shown for an 8-variable problem:

		variables							
depth	curvar	1	2	3	4	5	6	7	8
1	3	1	1		1	1	1		1
2	2	2			2	2	2		
3	5	3						1	2
4	1							2	
5	6							3	
6	7								
7	4								

"curvar" is variable assigned at depth k.

In this case, variable 1 for example, has one adjacent 'assigned' variable after the first variable selection, another after the second, another after the third, and none after that since it is the next variable selected at this point. Variable 8 has one adjacent 'assigned' variable after the first selection and another after the third (note that the only entries in the table occur where there are increments).

As a preliminary assessment, levels of search were chosen near the average faildepth for maximum forward degree, specifically levels 4 and 6, and the number of variables with 3-4 constraints with variables in the assigned set was tallied for each problem. Then a score was computed based on a weighted sum of these tallies, where variables with four constraints were weighted 1.6 times as much as variables with three constraints and variables with five constraints were weighted 2.5 times. The values of 3-4 constraints were chosen because with this number of constraints, the expected number of viable values is quite low (approximately two). The weights were based on ratios for expected number of viable values between 3 constraints and 4 or 5, respectively. Scores were then ranked in order to derive a Spearman Rank Correlation Coefficient between this measure and the measure of search performance.

In the next test, the measurements (taken at each step) were the size of the future-adjacent set, the number of connected components and the vertex connectivity of each component. Three levels of search were chosen for analysis: 2, 4, and 6. At these levels, all but 1-2 of the future-adjacent variables were in one large connected component (recall that fd was being used), so only this component was examined. The mean size of the large component was 24 at level 2, 33 at level 4, and 37 at level 6. Connectivity scores were also ranked before computing the correlation.

The third test was a refinement of the second. At each level of search, following the new assignment, variables in the future adjacent set were identified that were adjacent to $\geq k$ assigned variables. Here k was set to 3, for the reasons mentioned above. Then the same connected component, connectivity analysis described above was carried out for the subgraph that included these variables and their mutual constraints. In this case the measure was simply the number of variables in connected components of size greater than one. The idea was that with three or more adjacent singleton domains the number of viable values would usually be one or two, and in this case the likelihood of failure would be increase rapidly with the size of the subgraph.

The correlation with number of constraints with assigned varibles was 0.39 and 0.38 for levels 4 and 6. This is promising because it indicates some degree of relation between differences in performance and this variable. (In this case, correlations would not be expected to be high; see comments below.) Somewhat surprisingly, the correlations with connectivity at levels 2,4 and 6 were -0.07, 0.12, and 0.00. However, the correlation with connected subgraph size based on future k-adjacent variables was 0.37 for level 5 and 0.33 for level 6. The latter are still fairly low, but they do indicate an relation of the expected type. That is, they show that a simplification heuristic like fd is more effective in relation to a particular contention heuristic when the former produces larger future subgraphs composed of variables connected to larger numbers of assigned variables and when these variables also have more connections with each other.

In summary, although the results are in line with the analysis given in the last section, more work must be done (including a more elaborate analysis) before we can demonstrate clearly why a particular problem is more amenable to one or the other heuristic strategy. Currently, I am setting up to examine the degree patterns within the subgraphs formed the future-adjacent with large numbers of adjacent assigned variables. In addition, it would be useful to have data on the locations of wipeouts during arc consistency along the propagation paths in the future subgraph to determine the proportion of failures occurring in the future adjacent set.

4 Balancing Heuristic Actions

Previous work uncovered an important phenomenon that involves 'balancing' the effects of the two kinds of heuristic action, the so-called "heuristic synergy" effect [Wal06a]. To emphasize the striking nature of this effect, some data from the earlier paper are presented in the following table. These are selected cases from an experiment with six heuristics, in which all possible pairings were tested. The data for "compound" heuristics were obtained by rating the choices made by each heuristic (in this case with integers from 10 [best choice] down to 1) and then making the actual choice in terms of a weighted sum of these ratings. (Here the weights were equal.)

Note that when the synergy effect occurs, the results are usually well below the results for the *best* of the two heuristics when this is used by itself. Moreover, the effect can be seen (repeatedly) with individual problems, so it is not just a matter of matching the best of either alone on each problem. Now, the key point here is that in this experiment synergistic effects only occurred when the two heuristics favoured different factors; hence, using the knowledge of a heuristic's loading in the factor analysis, one could predict with 100% accuracy whether or not synergy would occur.

Even more striking effects can be demonstrated if forward checking is used instead of MAC as the basic algorithm. In this case, simplification heuristics perform very poorly. In the terms of the present discussion, this indicates that when simplification effects are limited to the constraints between the assigned and the future-adjacent variables, then strategies based on simplification perform poorly. Nonetheless, for appropriately weighted combinations (i.e. with contention weighted 3-5 times as much as simplification), the combined effects can outperform the individual contention heuristic by an order of magnitude.

Table 1. Predicted and Actual Synergies for Equal-Weight Pairs

heuristics	first	second	compound
dom+fd	11,334	2625	**1317**
dom+bkwd	11,334	27,391	12,521
dom+stdg	11,334	2000	**1427**
dom+dm/fd	11,334	1621	1890
fd+bkwd	2625	27,391	**1962**
fd+stdg	2625	2000	2344
fd+dm/dg	2625	2076	**1369**
dm/dg+dm/fd	2076	1621	1800

Mean nodes per problem. <50,10,0.184,0.37> problems. Italicised entries are predicted synergy based on factor loadings. Bold entries show actual synergistic effects. Based on Table 8 in [Wal06a].

Perhaps the most significant implication of this discovery is that it is an essential part of the explanation of why the various advanced heuristics proposed over the last 15 years outperform ordinary heuristics. The following table (taken from [Wal08]) shows the results of a factor analysis in which several advanced heuristics were included, including adaptive heuristics like impact and weighted degree. These experiments were based on means of 100 runs with random value selection for each problem. As indicated by the bold fonts, each of the advanced heuristics has a high loading ($\geq .6$) on *both* factors. (Results for these heuristics are shown under the line dividing the table.) Note also that both domain over degree heuristics show some balancing as well; this is to be expected given that their components are heuristics associated with different actions. What is interesting is that they do not show balancing effects to the same degree as advanced heuristics or with the same reliability.

It is also worth noting that performance differences among the advanced heuristics on this set of problems also seems to be related to the same effect. On these problems the best performance is obtained with the extended DVO algorithm and domain over weighted degree with random probing (labeled "probing" in the table). Evidence to date indicates that these heuristics balance the two heuristic factors more consistently than the others. (The evidence is based on factor analyses using one run with lexical value ordering; in this case, these two heuristics are the only ones that still show a definite balancing effect.)

Unfortunately, it not yet possible to present anything like a full account of the reasons for synergy. That is, the present analysis is not yet sufficiently refined to indicate why this strategy yields such striking results.

Table 2. Factor Analysis with Advanced Heuristics

heuristic	factor 1	factor 2	unique
dom	.458	.411	**.621**
d/dg	**.804**	.556	.044
d/fd	**.795**	.582	.028
fd	.434	**.875**	.045
dg*fd	.479	**.859**	.033
edgsm	.454	**.863**	.050
FF2	**.831**	.387	.160
FF3	**.822**	.429	.139
FF4	**.835**	.430	.117
prom	.393	**.712**	.339
stdg	.549	**.798**	.062
kappa	**.611**	**.770**	.033
DVO*1	**.706**	**.653**	.075
impact	**.660**	**.693**	.084
wdeg	**.717**	**.662**	.047
d/wdeg	**.789**	**.595**	.023
probing	**.706**	**.618**	.119

Notes. Based on 100 <50,10,0.18,0.37> problems. Values $\geq$ .60 shown in bold. Advanced heuristics are listed below the horizontal line. From Table 5 in [Wal08]

5 Relations between Analysis of Heuristic Actions and the Policy Framework

Prior to the work recounted here, the author was involved (together with Chris Beck and Pat Prosser) in an attempt to analyse variable ordering heuristics from a rather different perspective. This work led to the establishment of a "policy framework" for describing heuristic performance. The question is, how is this policy framework related to the present discussion of two basic heuristic actions? Roughly speaking, the difference between the two is a difference between what and why (or how).

The policy framework is based on a partition of the states of backtracking search into those in which search is on a path to a solution, i.e. the current partial assignment can be extended to a full consistent assignment, and those in which the present partial assignment cannot be so extended. Each of these classes of states is associated with an *ideal* policy, which if followed would minimise the number of search nodes in comparison with any other basis for selection. The first, called the *promise policy*, holds for states of the first type and says that one should make choices that maximise the likelihood of remaining on the solution path. The second, called the *fail-first policy*, holds for states of the second type and says that one should make choices that minimise the size of the unsolvable search tree (i.e minimise the size of the refutation with respect to the root of the unsolvable subtree). In and of themselves these policies are practically truisms; they become significant when they are associated with measures of adherence, which allow one to characterise heuristic performance in backtrack search with respect to the degree of adherence to each (ideal) policy. Using these measures, we can show,

(i) that heuristic quality is usually related to improvements in promise as well as fail-firstness [BPW03], (ii) that while adherence to both policies generally co-varies, there are exceptions, which can be demonstrated with these methods [BPW05,Wal06b]. This framework also allows us to characterise the well-known Fail-First Principle, which can be (re)defined as an (ideal) metaheuristic that says, always make choices as if the fail-first policy were in force. In other words, disregard the promise policy in making choices.

With this characterisation, it becomes clear that the Fail-First Principle is not an explanation (in the causal sense) at all. To say that a given heuristic outperforms, say, random variable selection because it follows the Fail-First Principle is essentially to say that it's effective because it behaves in an effective manner. This is like saying that search is reduced in extent because the search tree is smaller. In other words, the Fail-First Principle is a way of saying what a good heuristic has to do, but it doesn't explain why it is (or is not) successful in doing it.

Consider an example, which is highly interesting in itself (and which demonstrates the value of the policy framework). [SG98] first showed that a series of heuristics that are designed to show increasing degrees of fail-firstness did not show a commensurate ordering in terms of overall performance. (In terms of the present framework, increasing fail-firstness means increasing degree of adherence to the fail-first policy. Note also that their conception of fail-firstness is too limited since it only applies to failure across a specific set of constraints.) Although there was an unfortunate error in the original experiments, as shown by [BPW04], the latter authors did find a similar effect in that the heuristic FF3 was appreciably worse than FF2 although it was designed to have better fail-firstness (and actually does as verifed by [Wal06b] using the more adequate definition of fail-firstness). The latter authors also showed that this effect was only found when the forward checking algorithm was used – with MAC the ordering was that expected under the Fail-First Principle – but this is immaterial for the present argument. Using the policy framework, they were then able to show that the poor performance of FF3 is associated with a substantial fall-off in promise; given the completeness of the policy framework, this explanation is in that sense sufficient.

However, demonstrating an improvement in fail-firstness paired with a fall-off in promise does not explain why this happens. The reason seems to be that because of the formula that FF3 uses (shown below), it is possible to select a variable with a large domain if it is adjacent to several variables with small domains.

$$(1 - \prod_{v_j \in F, v_j constrains v_i} (1 - p_2^{m_j}))^{m_i}$$

Especially at the top of the search tree, this can have deleterious effects on performance, because although pruning is more effective (insoluble subtrees are smaller), there are many more insoluble subtrees to prune than if smaller domains had been chosen.

6 Heuristic Actions on Structured Problems

Earlier, it was conjectured that in the case of structured problems, the same basic heuristic actions would be effective and only these two, but that they might interact with the various structural features of the problem [Wal08] depending on the specific heuristic. Because of this, the same action might lead to differences in performance if it was associated with different structures, and as a result heuristics that behave similarly with homogeneous random problems might be much different with heterogeneous problems.

A simple example of this was already demonstrated in [Wal08], where it was shown that for the composed problems of [LBH04], heuristics in the FFx series ($x \geq 2$) behave differently than other contention heuristics. This is because these heuristics select variables in the highly constrained subproblem at the beginning of search, resulting in a different pattern of variation (and highly effective search).

The more significant question is the manner in which a heuristic action interacts with different forms of propagation, symmetry-breaking, etc. Currently this is being approached by studying problems with specific structural features, such as colouring problems (homogeneous constraints), relop problems (ordered domains and [some] ordered relations), and problems with random constraints and structured constraint graphs.

7 Conclusions

The analysis of CSP search necessarily has a dual aspect. Because constraint propagation is in many ways very well-behaved, it is possible to analyse many of its features through a kind of logical determinism. This is especially true at the level of individual relations. On the other hand, when relations are combined into networks, the sheer number of possibilities is such that some aspects of search can apparently only be analysed using probabilistic methods, which is the approach taken here.

The present analysis shows how inadequate it is rely on verbal "principles" that heuristics are supposed to adhere to (which generally have a beguiling surface plausibility). An example is the principle concerning variable selection enunciated by Refalo [Ref04]. This says that variables should be selected that maximally constrain the rest of the search space. However, if this were true without qualification, then simplification heuristics would always be better than contention heuristics. We have already seen that this is not so for random problems. These results hold for MAC; when the algorithm is forward checking, simplification heuristics are much *worse* than contention heuristics [Wal06a]. For other classes such as colouring problems, simplification heuristics are also two to three orders of magnitude worse than contention heuristics even on small (50-variable) problems using MAC [Wal08]. The reason is, basically, that in both these cases there is little interaction between variables in the future subproblem. With forward checking, this is because propagation is limited to the variables adjacent to the current variable. With colouring problems, this is because propagation rarely extends beyond the same point even with MAC, due to the nature of the constraints.

This research suggests new ways of designing heuristics based on problem assessment. Although this in itself is not new, to my knowledge this has never been done on the basis of first principles. Specifically, it should be possible to rapidly assess relevant parts of the future subgraph to decide whether a simplification or a contention heuristic is advisable at a given point in search. Unfortunately, although I think it would be of considerable theoretical interest, this method is likely to be overshadowed by the simple non-deterministic methods which balance these forms of heuristic action (see above). Another potentially useful application is the guidance of tie-breaking during randomised search.

In closing, I wish to add a thought on the nature of randomness, which may already be obvious to many. Considering the results on differential effects for problems from the same class of random problems, does this mean that random problems aren't really random? I think that what it really means is that here as elsewhere one must be careful in the way one defines something. In this case, the term "randomness" should be taken to refer to the ensemble of problems produced by some generating process, not to an individual problem within that ensemble. Individual random problems indeed have structure, as shown by this and a lot of earlier work, which can be exploited for better algorithm performance.

References

[BPW03] J. C. Beck, P. Prosser, and R. J. Wallace. Toward understanding variable ordering heuristics for constraint satisfaction problems. In *Proc. Fourteenth Irish Conference on AI & Cognitive Science-AICS'03*, pages 11–16, 2003.

[BPW04] J. C. Beck, P. Prosser, and R. J. Wallace. Failing first: An update. In *Proc. Sixteenth European Conference on Artificial Intelligence-ECAI'04*, pages 959–960. IOS, 2004.

[BPW05] J. C. Beck, P. Prosser, and R. J. Wallace. Trying again to fail-first. In B. Faltings, A. Petcu, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints-CSCLP 2004. LNAI No. 3419*, pages 41–55. Springer, 2005.

[HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[LBH04] C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Sixteenth International Conference on Tools with Artificial Intelligence-ICTAI'04*, pages 549–557. IEEE Press, 2004.

[Ref04] P. Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming-CP 2004. LNCS No. 3258*, pages 557–571. Springer, 2004.

[SG98] B. M. Smith and S. A. Grant. Trying harder to fail first. In *Proc. Thirteenth European Conference on Artificial Intelligence-ECAI'98*, pages 249–253. Wiley, 1998.

[vB06] P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 85–134. Elsevier, Amsterdam, 2006.

[Wal05] R. J. Wallace. Factor analytic studies of CSP heuristics. In *Principles and Practice of Constraint Programming-CP'05. LNCS No. 3709*, pages 712–726. Springer, 2005.

[Wal06a] R. J. Wallace. Analysis of heuristic synergies. In B. Hnich, M. Carlsson, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints-CSCLP 2005. LNAI. No. 3978*, pages 73–87. Springer, 2006.

[Wal06b] R. J. Wallace. Heuristic policy analysis and efficiency assessment in constraint satisfaction search. In *Proc. Eighteenth International Conference on Tools with Artificial Intelligence - ICTAI'06*, pages 305–312. IEEE Press, 2006.

[Wal08] R. J. Wallace. Determining the principles underlying performance variation in CSP heuristics. *International Journal on Artificial Intelligence Tools*, 17(5):857–880, 2008.

Using Constraint Programming to Perform Life Cycle Inventory Analysis Based on the Matrix Model

Richard J. Wallace[1], Colin Jury[2], Antonino Marvuglia[2], and Enrico Benetto[2]

[1]Cork Constraint Computation Centre and Department of Computer Science
Western Gateway Building, University College Cork, Ireland
[2]Resource Centre for Environmental Technologies, Public Research Centre Henri Tudor
66, rue de Luxembourg P.O. Box 144, L-4002 Esch-sur-Alzette, Luxembourg
r.wallace@4c.ucc.ie, {colin.jury,antonino.marvuglia,enrico.benetto}@tudor.lu

Abstract. Life cycle assessment (LCA) is an important approach to evaluating environmental impacts of products throughout their life-cycles. The basic formal apparatus for representing economic flows and the resulting environmental burdens is as a set of linear equations that can be handled with the tools of matrix algebra. In this paper we introduce a new linear programming (LP) model for life cycle assessement (LCA), which, unlike previous LP formulations, is based on and in some respects is equivalent to the standard matrix model. By using an LP model of this form, we retain the basic formal methodology for LCA, while setting the stage for incorporating the standard analysis into many decision making problems. This approach also obviates the need for special techniques such as those required to handle rectangular matrices, although the straight-matrix-calculation approach must be replaced by search if a specific value for the functional unit is required. More importantly, when LCA is modeled in this way it can be readily incorporated into more elaborate constrained optimisation problems, as a kind of soft global constraint. We consider some examples, modeled as constraint satisfaction problems, as well as an application of this approach to a particular real-world life cycle assessment study.

1 Introduction

Every product passes through a cycle of design, production, use, and disposal. And each stage in this cycle is a process that has wider effects. The purpose of life cycle assessment (LCA) is to gauge these effects from the perspective of the entire cycle ("cradle-to-grave"), so that overall impacts can be accurately estimated and tradeoffs recognized, in particular the "shifting of a potential environmental burden between life cycle stages or individual processes" [FIT$^+$06]. If done properly, it should enable the various stakeholders to make wiser decisions in regard to each of the several stages of a product's cycle. Obviously, for this to be done properly one must have tools that allow such summary accounts to be derived.

During the past 20 years methods have evolved for carrying out such analyses. Formally, the standard mathematical formulation for LCA (the *matrix method*) is based on matrices whose entries represent inflows or outflows of commodities (energy and materials) to and from each of the processes involved in the life cycle of the product investigated. Because these flows in turn can be associated with environmental burdens, matrix calculations can be used to derive the latter in a fairly straightforward fashion.

At the same time, the model in its simplest form has some serious limitations. One is the assumption of a 1:1 relation between production processes and economic flows. This results from the fact that the original matrix of economic flows must be inverted in the process of deriving the vector of environmental flows, or burdens. A more general problem is that the model doesn't allow straightforward testing of the properties required by a production process given specified limits on the environmental burdens.

In this work, we present an alternative approach based on linear programming. This approach has the advantage that it deals with the process matrix as a set of linear equations. Since in this case there is no need for matrix inversion, the problem of inverting a non-square matrix does not arise. We show that the same result is obtained with this model as with the standard matrix calculations, when the environmental budens are equal.

Although optimisation methods have been proposed for LCA in the past, and are also based on linear programming models [AC99b,FTF01], this was not done in a way that is consistent with the matrix model. Nor would this have been possible given the way these models were set up. Our approach, in contrast, can serve as an alternative means for deriving economic flows consistent with a given set of environmental burdens. Basically, we take the ordinary matrix model and calculate backward from a given set of burdens to the (best) possible economic demand rather than calculating forward from the demand to associated burdens.

An even more important benefit of this approach is that it can be nested inside larger problem representations, which are often necessary in areas such as product design, where different processes associated with different designs must be compared. Life cycle assessment then becomes a specialised component of the model; in fact, it can be viewed as a kind of global constraint, where LP methods are used rather than constraint propagation. In some cases, this provides an alternative to non-linear programming approaches.

In the next section we give a brief overview of LCA and the original method used for this purpose. In Section 3 we briefly describe the basic formalism of the matrix model. In Section 4 we describe some important examples of mathematical programming models that have been proposed in the past. Section 5 gives a detailed account of our LP model. Section 6 considers extensions, some of which may be fruitfully carried out with techniques from constraint programming, and Section 7 describes our initial implementation. Section 8 describes an example based on an LCA currently being carried out in an real-world industrial context and ends with some conclusions and prospects.

2 Background and Overview

The LCA methodology has been standardised by the International Standards Organisation (ISO) [ISO98]. In the ISO framework, a life cycle runs from raw material acquisition through product disposal. In the original framework, social and economic impacts were considered outside the scope of LCA, which was restricted to environmental concerns. However, these other impacts can be treated by extending LCA, e.g., to social LCA [BM10] and Life Cycle Costing [HLR08,Kro08].

LCA conceives of the product life cycle in a systems-theoretic way, as a set of processes leading to a given functional output. As such, it is function-oriented rather than product- or materials-oriented, which makes it suitable for comparing and redesigning products and processes [BHvdV$^+$00]. Both the functional output and the process (or *reference*) flows are defined in terms of units; as a result, the processes of the system are considered in relation to a *functional unit*. To make this more concrete, we employ the illustration given in [ISO98]:

> In the function of drying hands, both a paper towel and an air-dryer system are studied. The selected functional unit may be expressed in terms of the identical number of pairs of hands dried for both systems. For each system, it is possible to determine the reference flow, e.g. the average mass of paper or the average volume of hot air required for one pair of hand-dry, respectively. For both systems, it is possible to compile an inventory of inputs and outputs on the basis of the reference flows. At its simplest level, in the case of paper towel, this would be related to the paper consumed. In the case of the air-dryer, this would be related to the mass of hot air needed to dry the hands.

A major benefit of the ISO standard is that it sets up a well-defined division of labour in regard to the various tasks involved in assessing environmental effects. Three stages of LCA are defined: goal and scope definition, life cycle inventory analysis, and environmental impact analysis. All stages, in addition, involve "interpretation"; this is a shorthand (and perhaps not entirely satisfactory) way of referring to the continuous meta-assessment that is required in an actual application of LCA, and the series of adjustments or iterations that ensue. Thanks to this modularised schema, we are able to focus on one stage in (conceptual) isolation from the rest. Here, we restrict our attention to the life cycle inventory analysis (LCI). In this stage, having already established the boundaries of the system, the functional unit, and the unit processes and flows, we must quantify the inputs and outputs in terms of the functional unit. This includes assessment of those outputs of the unit processes (elementary flows in ISO terminology) that constitute *environmental burdens*.

Two widely used methods for performing LCI are the process flow diagram (or *sequential*) method and the matrix method [SH05]. The first method represents the original approach to modeling LCA. The processes involved in the production, consumption, and disposal of a product are represented as a flow chart, and for each step in the process the associated production of some environmental emission (e.g. CO_2) is determined. A hypothetical example is shown in Figure 1 based on the example from [ISO98] given earlier. (Note. This figure was devised by the authors and is not part of the original document.) The total environmental burden for the given output is obtained by summing across processes.

Although the process flow model is straightforward and easy to understand, it is difficult to formalise with mathematical notation and it has some fundamental shortcomings [HS02]. For example, feedback loops (such as the use of fuel to produce electricity and the use of electricity to produce fuel) must be solved by interrupting the feedback loop after a specified number of iterations or at the point when the change in demand is less than some specified amount, replacing the process data by data that accounts for

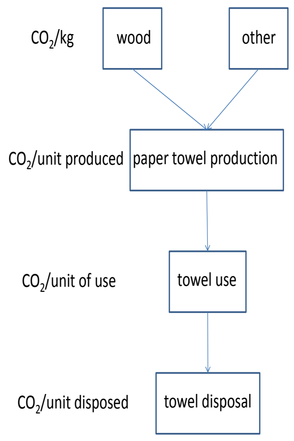

Fig. 1. Illustrative example of process flow model, based on example in [ISO98].

the feedback loop, or using an infinite geometric progression. In addition, it does not directly convey the global perspective that is the focus of LCA. One consequence of the latter is that the process flow model tends to obscure the formal structure of LCA. For these reasons, a second approach was developed during the 1990's, which is described in the next section.

3 The Matrix Model

The matrix method for the solution of the life cycle inventory problem is based on a set of linear equations (called "balance equations"). The left sides of the equations, which represent economic activities and environmental effects, are divided into two parts, the *technology matrix* (A) and the *environmental intervention matrix* (B). Correspondingly, the right sides of the equations are separated into two vectors, the *external demand vector* (f) and the *inventory table* (g). The basic matrix equations are then:

$$As = f \tag{1}$$

and

$$Bs = g \tag{2}$$

The vector s, which makes the left and right sides equal is called the *scaling vector*; critically, this same vector of scaling factors is used both to calculate demand and to calculate the vector of environmental burdens, g.

The strategy for solving this system of equations is to first solve for s in equation (1), according to the following equation,

$$s = A^{-1}f, \tag{3}$$

which involves inverting the matrix A. Then s can be used in equation (2) to obtain the vector of environmental burdens, g, associated with the life cycle of the investigated product. From this vector, various environmental impacts can be derived using standard equations.

Values in the A and the B matrices (as well as the g vector) can be of either sign. In the A matrix a negative sign means that the flow is an input for the process in question, while a positive sign represents an output (product). In the B matrix and the inventory table a positive sign indicates that the burden is emitted into the environment, while a negative sign represents natural resources extracted from the environment.

An appealing feature of this model is that it affords a clean separation between economic activities and environmental consequences while linking them together in a compelling and rigorous fashion. It is also more clearly related to the concept of a functional unit than is the process flow model.

3.1 Allocation Issues

The term "allocation" originally referred to several general issues concerning system boundaries and the problem of relating (allocating) environmental burdens to specific economic processes [Hup94]. Within the formalism of the matrix model, it refers primarily to cases where some production processes are multi-functional [Hei98,HS02], and in this paper the term is only used in this restricted sense. For example, two chemicals might be produced in the same production process, or a process may produce both heat and electricity together, and it may be possible to use both outputs. Various forms of recycling of outputs can also be brought under the same heading, see [Hei98,HS02].

For the matrix method, a problem arises when the economic matrix is no longer square because there are more flows than processes. This means that the standard matrix inversion technique just described cannot be applied to the problem in its original form. Various methods have been proposed for handling this situation. In some cases this involves adding extra rows, sometimes also using additional coefficients to split a flow from the original process into two or more flows. Other methods are based on the pseudo-inverse matrix, or on least squares techniques (ordinary, data, or total least squares methods) which are used to find a best fit [MCH10].

4 Mathematical Programming Models for LCA

Here, we briefly review work that has used some notion of optimisation in this context. All the models that we are aware of are linear programming (LP) models.

In [AC94], an LP model is presented for maximising profitability given various capacity constraints on the physical production system. This is, therefore, a cradle-to-gate model. Given some optimal profit, then environmental burdens can be calculated from the inputs and outputs associated with this value for the objective.

[AC98] describe an LP model where an objective can be defined either in terms of economic goals or in terms of environmental burdens (inventory levels). In the latter case, the goal is to minimize an aggregate function over these burdens, i.e.

$$\min \sum_{i=1}^{n} b_{ji} x_i,$$

"where $b_{j,i}$ is the burden j from process or activity x_j" (p. 307). This is carried out under various constraints such as balance constraints between different processes, economic demands, raw material supply, and constraints on production capacity. If an economic objective is used, the model is the same as in [AC94].

According to the authors' discussion, the intention here is to incorporate more of the internal structure of the production process into the model than is possible with the matrix technique. As such, this type of LP model seems more closely related to the sequential method; in addition, it is more in line with a cradle-to-gate than a cradle-to-grave analysis, which is the basic perspective of LCA.

One further issue with this particular model is that the basis for aggegrating different environmental burdens is not made clear. That is, a common currency is not specified. From the discussion in the Appendix to [AC98], it appears that only one kind of burden was being considered, e.g. CO_2, as a function of the different activities, which obviates the problem of a common currency.

Although a multi-objective orientation is implicit in [AC98], the approach is still piecemeal. In subsequent work, this orientation is made more explicit and thoroughgoing by the adoption of a multi-objective programming model [AC99b,AC99a]. In this way, they finesse the common currency problem just noted: different environmental burdens are now associated with different points on a Pareto frontier, and it is a subsequent (multi-criteria) decision-making problem to select one of these non-dominated points.

Freire et al. have developed an LP approach inspired by economic input-output analysis, which they call "life cycle activity analysis" [FTF01]. In this model, environmental burdens are considered to be additional outputs of the system. The basic I/O matrices are then partitioned according to whether the output consists of intermediate products, final products, or environmental outputs. Clearly, this is essentially the same as the partition of the matrix model into the technology and environmental matrices. Commensurate with the I/O framework, the objective is to minimise the overall cost related to economic activities. This is subject to constraints on supplies, demands, and environmental burdens. (In the same vein as [AC98], a variant of their model is presented for handling environmental impacts; in this case, however, constraints on both burdens and impacts are included. [The motivation for this is unclear.])

Although based on a different approach to LCA, this work is quite similar to the present work in its basic intentions and strategies. Perhaps the most significant difference is that this model is not based on the idea of a functional unit and so does not fall within the standard framework for LCA. Moreover, demand is treated as a constraint on final production, to ensure that these values are greater or equal to some given minimum. In addition, environmental burdens are all positive, i.e. there are no burdens that represent losses of environmentally relevant materials. (To some extent, these are represented through constraints on supplies.) But despite these differences, this work is of potential relevance to the present efforts.

Discussions of allocation are found in some of the papers on linear programming discussed here, e.g. [AC94,AC98]. However, since the problem is not described in terms of the matrix model, these discussions are tangential to the concerns of the present paper.

Finally, it should be noted that, despite the variety of operations-research-based approaches to evaluating environmental impacts of production processes that have been proposed (see [FTF01] for further examples), to our knowledge there have been no cases where constraint programming techniques have been applied in this domain.

5 An LP Version of the Matrix Model

As noted by [AC99b], LCA is presently "based on linear models of human economic activities and the environment" (p. 137), and therefore the basic relations can be readily expressed as an LP.

In our model, the objective is to maximise the demand, f, given constraints that limit this vector to having one non-zero value in accordance with the usual matrix model; it is trivial to add these to the LP model. This is subject to the constraints represented by both equations (1) and (2). However, while the linear equations included in (1) are represented by equality constraints as in the original equation, those included in (2) become inequality constraints ($\leq$ if the inventory table value is positive; $\geq$ if the value is negative). Note that in this model we do not have to derive an inverse matrix A^{-1}. Instead, the balance equations in (1) and (2) together are used to derive values for s and f. The model is shown in Figure 2. In this case, the model is presented in a form that reflects the basic logic of the approach. For example, the equation $As = f$, which is carried over from the basic matrix model, can be changed to standard LP form by subtracting f from both sides.

Perhaps the most distinctive aspect of this model is that the values of s are treated as variables (i.e. activities) although they are not effective components of the objective (i.e. they have coefficients of 0). It should also be noted that when a value of g is negative, then this is also true for at least one value (b_{ij}) on the lefthand side of that relation, and this guarantees that some positive value of s will satisfy the relation. Finally, although not necessary, it is sometimes convenient to specify a bounding value for f_k, i.e. to specify a maximum reference flow. Otherwise, f_k is bounded by the relevant linear equation in $As - f = 0$, since in a well-formed matrix $\sum a_{kj} s_j$ will always be a positive number. This is because it represents an output of the same form as f_k.

In this LP model, as in the matrix model, the economic and environmental flows represented by the A and B matrices are givens obtained from some environmental database. In contrast to the usual methods based on the matrix model, we also stipulate a set of values for the inventory table, g, that serve as resource constraints. In other words, these are values that constrain the environmental burdens. We then obtain a value for the demand, which is the optimum (maximum) given these constraints. This value is directly related to the value associated with these inventory table values in the usual matrix calculations, as indicated by the following proposition. (Here we assume that the matrix is square and non-singular, i.e. that it has an inverse.)

```
Find values for f and s such that:
    $Z = \sum f_i$ is maximised
subject to the following constraints:
    $As = f$
    forall (i in inventory components)
      if $g_i \geq 0$
        $\sum b_{ij} s_j \leq g_i$
      else
        $\sum b_{ij} s_j \geq g_i$
    forall (i in demand components)
      if $i \neq k$
        $f_i = 0$
```

Fig. 2. LP model based on LCA matrix model. A and B are the technology and intervention matrices, s the scaling factor, g the inventory table. f_k is the reference flow.

Proposition. *Given a vector of environmental burdens, the value for demand obtained from the LP model described above is identical to the demand from which these burdens are derived by the matrix inversion method.*

Proof. Given a set of values for the A and B matrices, and a specified demand, there is only one set of values for the scaling vector and, therefore, a single solution (set of values) for the inventory vector. Conversely, given this same set of values for the inventory vector, suppose the value for the functional output were lower than the functional unit. This would result in a different set of values for the scaling factor. But in this case given that B is the same, the inventory vector would be different. □

Since the present method is based on linear constraints, it is both efficient and scalable. This means that we can use it repeatedly to answer queries regarding alternatives in the manner described above. In this way we can determine a set of environmental burdens that are consistent with a certain value for the demand.

Another significant feature is that the present model incorporates *all* of the features of the matrix model, as described in [HS02]. In particular, the conventions pertaining to negative and positive values in the A and B matrices can be retained. This is the reason for the select statement in the model depicted in Figure 2; burdens that reflect consumption of natural resources are associated with $\geq$ constraints since here one wants to minimise such burdens. Burdens that reflect outputs such as CO_2 are minimised via $\leq$ constraints.

One question that might be raised regarding this LP-based approach to LCA is whether this affects the concept of a functional unit, since now this unit is no longer a “reference flow” on which we base our analysis of environmental impacts. However, since in the usual matrix method the reference flow is chosen arbitrarily, this is not a serious deviation from the normal use of the model. In other words, we do not violate the basic logic of the model; we simply use it in a somewhat different fashion. Moreover, as previously noted we can set a bound (on f_k) equal to the functional unit and determine whether this value can be achieved. We may also be able to adjust flows in the model to make this possible (see Case Study below).

The present model can also be used for multifunctional processes in conjunction with standard methods for dealing with this situation within the matrix-model framework. These include the substitution and partition methods (cf. [HS02]. The proposition on equivalence under equal burdens also holds by the same arguments given before.

6 Extensions to LP/CP hybrid models

The LP model that we have described can be embedded in more complicated models that have greater flexibility as well as additional ‘side’ constraints.

For example, in designing a product for a given purpose, often one would like to set limits on the flow for a given (abstract) production process, which is consistent with the functional unit and with a set of environmental burdens. Given such limits, one can then try to design an actual process that satisfies these limits. In this way, the matrix model becomes a discovery tool.

For this purpose, the present model can be extended in either of two ways. The first is to turn it into a nonlinear program, where the matrix of coefficients, A, becomes a set of variables together with the scaling factor s. In other words, equation (1) becomes a set of nonlinear equations. An alternative approach, which we are pursuing, is to retain the LP whilst embedding it in a larger constraint program. One strategy is to consider a feasible range of values for some coefficients (which are rates of flow), and to divide the range into subranges, each associated with an integer value. The latter are then the domains of a set of auxiliary variables X_a. To select a value for coefficient a_{ij} the current value of the associated auxiliary variable $X_{a_{ij}}$ is multiplied by some unit subrange. Greater precision can be obtained by increasing the size of the domain of $X_{a_{ij}}$ (and decreasing size of the unit subrange appropriately).

In this case, search for an optimal solution can be carried out with a branch and bound process in which different values for certain coefficients are tried in order to obtain the maximal demand under a set of limiting values for the environmental burdens. This method is feasible because in most cases only a few of the coefficients of A need to vary (see Case Study below). With this approach, we can also readily incorporate further side constraints, such as constraints between different unit flows (again, cf. Case Study).

From the perspective of constraint programming, the LP model can be considered a kind of (soft) global constraint, i.e. an LCA constraint. Conceptually, as well as operationally, this is a very appealing formulation. More generally, this approach seems well-suited to sensitivity analyses and the like. In [ISO98] it is emphasized that in practice LCA is a highly iterative process. The present methods would seem particularly well-adapted to these circumstances.

7 Implementation

The basic LP model as well as a simple CP/LP hybrid have been implemented in ILOG OPL Studio, an optimisation modelling system for mathematical programming and constraint programming [Hen99]. The OPL LP model follows the schema in Figure 2 very closely. The only difference is that, because we were not able to use the select features in OPL in the way depicted in this figure, we replaced the original inventory table with two tables, one with positive and one with negative values. For each of these new vectors, values not in the original vector were given (+/-)BIGNUM values so that the associated constraints were always satisfied. In addition, a hybrid model has been implemented using the script feature of OPL. This allows us to call the LP model repeatedly after certain economic flows are set, at the same time satisfying some constraints between them.

8 A Case Study

A preliminary study has been made of this constraint-based approach using an LCA obtained from the Public Research Centre Henri Tudor. This application concerns the energy valorisation of grape marc by the production of grape marc pellets. Data about the different steps involved in the process were obtained from a survey accomplished at the production site, in a Luxembourgish factory. The chosen functional unit is 1000 MJ of heat produced by the production and the combustion of pellets compliant with the new standards for solid biofuels within Europe [BSI10].

The technology matrix of the full system comprises over 1,600 entries, and the inventory vector has over 1,500 elements, of which only a few have been selected to test the optimisation algorithm. In the LCA already carried out at the Centre [JK11], the allocation among different co-products (heat from pellets; press juice produced during the grape marc dehydration phase and ashes coming from the combustion of pellets) was tackled using both the system expansion and cut-off methods (cf. [HS02]), but the above described LP has been applied only to the linear system obtained with the latter. The results of that study showed that the water content in fresh grape marc entering the dehydration phase has a significant influence on the various impact indicators, mainly because it affects the energy consumption required and the composition (and, as a consequence, the putrescibility) of the press juice, which is subsequently digested in order to produce biogas.

Preliminary tests have been carried out with a reduced (14 X 14) matrix. The g vector was limited to three environmental burdens: total carbon dioxide, total carbon monoxide, and total methane. (In the table below, their values appear in this order.) For this problem, a given LP is solved almost ‘instantly’ with OPL Studio for any set of environmental burdens. In the present problem there are seven entries in the (reduced) technology matrix that depend on the water content in fresh grape marc, here denoted by x. In each case, this dependency can be represented by a linear equation of the form

$y_i = a_i x + b_i$. Hence, in the CP representation there is only one additional variable that represents x. Since smaller values of x yield higher values for the optimal demand (because most of the affected flows increase in absolute value when x decreases), a binary search strategy could be used to locate the value of x (to any level of precision) for which the associated flows gave a greatest demand $\leq$ the functional unit. In some cases there was a range of values of x that gave an optimal value for demand that was equal to the functional unit. In this case, the highest value for x is the greatest value consistent with the functional unit, which gives us a limit for the acceptable water content. Typical results are shown in the following table. For each run there were 18 calls to the LP solver; in the table below, time is based on a running sum updated after each call.

burdens	greatest demand	highest x value	time
(5,5,5)	603	–	.0000
(10,10,10)	1000	9.09	.0000
(20,20,20)	1000	44.99	.0000

This small case study serves to demonstrate the viability of the present approach. It also gives some indication of the potential power and generality of a constraint-based approach for handling sustainability issues involving life cycle assessment. Given these promising results, we can begin to explore other scenarios where constraints of various kinds are added to the basic LCA problem.

References

[AC94] A. Azapagic and R. Clift. Allocation of environmental burdens by whole-system modelling – the use of linear programming. In G. Huppes and F. Schneider, editors, *Proceedings of the European Workshop on Allocation in LCA*, pages 54–60. SETAC-Europe, Leiden, 1994.

[AC98] A. Azapagic and R. Clift. Linear programming as a tool in life cycle assessment. *International Journal of Life Cycle Assessment*, 3:305–316, 1998.

[AC99a] A. Azapagic and R. Clift. The application of life cycle assessment to process optimisation. *Computers and Chemical Engineering*, 23:1509–1526, 1999.

[AC99b] A. Azapagic and R. Clift. Life cycle assessment and multiobjective optimisation. *Journal of Cleaner Production*, 7:135–143, 1999.

[BHvdV$^+$00] M. Bouman, R. Heijungs, E. van der Voet, J. C. J. M. van den Bergh, and G. Huppes. Material flows and economic models: An analytical comparison of SFA, LCA and partial equilibrium models. *Ecological Economics*, 32:195–216, 2000.

[BM10] C. Benoit and B. Mazin, editors. *Guidelines for Social Life Cycle Assessment of Products*. UNEP, 2010.

[BSI10] BSI. *Solid Biofuels. Fuel Specifications and Classes - Part 1: General Requirements*. BS EN 14961-1. British Standards Institution, 2010.

[FIT$^+$06] M. Finkbeiner, A. Inaba, R. B. H. Tan, K. Christiansen, and H.-J. Klüppel. The new international standards for life cycle assessment: ISO 14040 and ISO 14044. *International Journal of Life Cycle Assessment*, 11:80–85, 2006.

[FTF01] F. Freire, S. Thore, and P. Ferrão. Life cycle activity analysis: Logistics and environmental policies for bottled water in portugal. *OR Spektrum*, 23:159–182, 2001.

[Hei98] R. Heijungs. A special view on the nature of the allocation problem. *International Journal of Life Cycle Assessment*, 3:321–332, 1998.

[Hen99] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT, Cambridge, MA, 1999.

[HLR08] D. Hunkeler, K. Lichtenvort, and G. Rebitzer, editors. *Environmental Life Cycle Costing*. SETAC Press, Pensacola, FL, 2008.

[HS02] R. Heijungs and S. Suh. *The Computational Structure of Life Cycle Assessment*. Kluwer Academic, Dordrecht, The Netherlands, 2002.

[Hup94] G. Huppes. The issue of allocation. In G. Huppes and F. Schneider, editors, *Proceedings of the European Workshop on Allocation in LCA*, pages 1–2. SETAC-Europe, Leiden, 1994.

[ISO98] ISO. *14040. Environmental Management – Life Cycle Assessment – Principles and Framework*. International Organisation for Standardisation, Geneva, Switzerland, 1998.

[JK11] C. Jury and G. Kniep. *Energy Recovery of Biodegradable Waste Streams from Wine Production (MARC)*. Report WP3.1, 3.2: Scenario Definition & Evaluation. CRP Henri Tudor, Esch-sur-Alzette, Luxembourg, 2011.

[Kro08] Y. Krozer. Life cycle costing for innovations in product chains. *Journal of Cleaner Production*, 16:310–321, 2008.

[MCH10] A. Marvuglia, M. Cellura, and R. Heijungs. Toward a solution of allocation in life cycle inventories: The use of least-squares techniques. *International Journal of Life Cycle Assessment*, 15:1020–1040, 2010.

[SH05] S. Suh and G. Huppes. Methods for life cycle inventory of a product. *Journal of Cleaner Production*, 13:687–697, 2005.

Author Index